Worst-case execution time analysis for a Java processor

Schoeberl, Martin; Puffitsch, Wolfgang; Pedersen, Rasmus Ulslev; Huber, Benedikt

Published in:
Software: Practice & Experience

Link to article, DOI:
10.1002/spe.968

Publication date:
2010

Document Version
Early version, also known as pre-print

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **0**:1 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Worst-case execution time analysis for a Java processor

Martin Schoeberl,*[1] Wolfgang Puffitsch,[1] Rasmus Ulslev Pedersen,[2] and Benedikt Huber[1]

[1] *Institute of Computer Engineering, Vienna University of Technology, Austria*
[2] *Department of Informatics, Copenhagen Business School, Denmark*

SUMMARY

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 at Java bytecode level. We present a Java processor, called JOP, designed for time-predictable execution of real-time tasks. The execution time of bytecodes, the instructions of the Java virtual machine, is known cycle accurately for JOP. Therefore, JOP simplifies the low-level WCET analysis. A method cache, which fills whole Java methods into the cache, simplifies cache analysis.

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. An integrated data-flow analysis performs receiver type analysis for dynamic method dispatches and loop bound analysis. Furthermore, a model checking approach to WCET analysis is presented where the method cache can be exactly simulated.

The combination of the time-predictable Java processor and the WCET analysis tool is evaluated with standard WCET benchmarks and three real-time applications. The WCET friendly architecture of JOP and the integrated method cache analysis yield tight WCET bounds. Comparing the exact, but expensive, model checking based analysis of the method cache with the static approach demonstrates that the static approximation of the method cache is sufficiently tight for practical purposes.

KEY WORDS: Worst-case execution time; Java processor; Real-time system

1. Introduction

The Java programming language is a safe and 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 [10]. Worst-case execution time (WCET) analysis is of primary importance for hard real-time systems. In this paper, we present a complete solution with a time-predictable Java processor and a supporting WCET analysis tool (WCA).

*Correspondence to: Martin Schoeberl, Institute of Computer Engineering, Vienna University of Technology, Austria
E-mail: mschoebe@mail.tuwien.ac.at

Received 14 May 2009
Revised 8 December 2009

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 tools is the growing complexity of new processors. It is almost impossible to model them for the low-level analysis. Even if the processor can be modeled, the analysis with the resulting model is too expensive to be feasible.

We address this problem by a processor architecture designed to enable simple and efficient low-level analysis. The Java processor JOP [62, 64] provides time predictable execution of Java bytecodes and a real-time instruction cache [59]. JOP is open-source under the GNU GPL. Therefore, all sources, including the processor description in VHDL, are freely available. The VHDL description documents the architecture. Therefore, the timing properties of JOP can be verified. In this paper, we present a WCET analysis tool based on the implicit path enumeration technique (IPET) and a variant based on model checking. We compare the strengths and weaknesses of both techniques. The analysis of JOP's method cache is integrated in the presented tool. Furthermore, the tool includes data-flow analysis for automatic loop bound detection and receiver type analysis. The approach is evaluated by analyzing several real-time benchmarks and applications and comparing the resulting WCET with the execution time on the Java processor.

The paper is structured as follows: The remainder of this section describes Java in the context of safety-critical applications and the contributions of the paper. Section 2 discusses related work in the field of Java processors, WCET and data-flow analysis, and WCET analysis for Java and JOP. Section 3 gives a brief overview of the Java processor JOP. Section 4 presents WCET analysis based two different approaches: (1) implicit path enumeration and (2) model checking. The integration of both methods into our WCET analysis tool is described. Furthermore, the low-level analysis for JOP is given. The approach to analyze the instruction cache is given in Section 5. In Section 6 the implemented data-flow analysis for loop bounds and receiver types is explained. The combination of the real-time Java processor and the WCET analysis tool is evaluated in Section 7. The findings and further paths to explore are discussed in Section 8. Section 9 concludes the paper.

1.1. Java for Safety-Critical Applications

Java is considered as a replacement of Ada for future safety-critical applications [76]. A Java Community Process developed standard, JSR 302: Safety Critical Java (SCJ) Technology [30], is expected to be finalized 2010. SCJ is based on the execution model of the Ravenscar profile for Ada [14].

A SCJ application consists of a non-real-time initialization phase and a mission phase. During the initialization phase, all schedulable objects are created and data structures for communication are allocated. During the mission phase the time-triggered and event triggered handlers (the schedulable objects) are scheduled by a preemptive, fixed priority scheduler. Object allocation is only allowed for temporary data objects in a private scoped memory. SCJ defines three levels: Level 0 represents a cyclic executive with a single thread of control; level 1 is a single mission with periodic and aperiodic event handlers, and level 2 supports nested missions and RTSJ style real-time threads.

We consider our proposed combination of a time-predictable Java processor and WCET analysis at Java bytecode level in the context of safety-critical Java. The execution model of handlers, which are

invoked periodically or in response to an event, simplifies the WCET analysis. Avoidance of voluntary waiting for the next release, as possible with waitForNextPeriod() in the RTSJ, simplifies the timing analysis. Furthermore no blocking, which defeats WCET analysis, is allowed in level 0 and 1 of SCJ. SCJ supports a memory model with scoped memory that does not need a garbage collector. Therefore, no WCET analysis of a collector is needed.

1.2. Contributions

This paper presents a complete solution for time-predictable execution of real-time Java tasks: a time-predictable Java processor and the accompanying WCET analysis framework. The open-source approach enables independent verification of the presented results and allows other researchers to extend the tools.

This paper is an extension of earlier work [68]. Besides a redesign of the WCET tool WCA, the paper and the tool are enhanced by following contributions:

1. Data-flow analysis (DFA) is integrated into the WCET analysis tool. DFA extracts simple loop bounds and many source code annotations can be avoided. DFA also performs receiver type analysis to tighten the WCET on dynamic method dispatch.
2. Model checking for WCET analysis is evaluated and compared with the IPET based analysis. The presented tool integrates both approaches and tradeoffs between analysis time and accuracy are presented. To the best of our knowledge, this is the first presentation of a direct comparison between model checking and IPET based WCET analysis.
3. A static analysis of the method cache is integrated into the IPET based analysis; an exact method cache analysis is performed with model checking.
4. The timing information for individual Java bytecodes is automatically extracted from the microcode of the Java processor.
5. The paper contains an extended evaluation section. In addition to three real-world applications, Java versions of the Mälardalen benchmarks [43, 25] are used.

2. Related Work

As the paper covers the combination of a time-predictable Java processor and WCET analysis methods/tools, the following section discusses related work on Java processors, general WCET and data-flow analysis techniques, WCET analysis for Java and available WCET tools that target JOP.

2.1. Java Processors

Sun introduced the first version of picoJava [45] 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 [51]. Compared with picoJava, JOP is less resource demanding and can be implemented in about 3.5K logic cells.

aJile's JEMCore is a Java processor that is available as both an IP core and a stand alone processor [21]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. The Cjip processor [20, 34] 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 on the Cjip. Komodo [36] 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. jamuth [75] is the commercial version of the Java processor Komodo. SHAP [79] is a research Java processor based on the architecture of JOP and enhanced with a hardware object manager. SHAP also implements the method cache [50].

For none of the described Java processors the timing information is available.[†] 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 [61].

2.2. WCET Analysis

Shaw presents in [69] 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 [56] and [39]. 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 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. A conditional branch usually has a different execution time, depending on whether it is taken or not. This difference is represented by two edges with different weights.

In [39] a similar approach with ILP is proposed. However, they use the CFG as the basis to build the ILP problem. The approach is extended to model the instruction cache with cache conflict graphs [40].

WCET analysis of object-oriented languages is presented by Gustafsson [18]. Gustafsson uses abstract interpretation for automatic flow analysis to find loop bounds and infeasible paths. The work is extended to a modular WCET tool with instruction cache analysis and pipeline analysis [15].

Cache memories for the instructions and data are classic examples of the paradigm: *Make the common case fast*. Avoiding or ignoring this feature in real-time systems, due to its unpredictable

[†]We tried hard to get information for the aJile processor. For the Cjip timing information is published, but not accurate [62]. We are in contact with the developer of jamuth for the instruction timing of jamuth.

[‡]A basic block is a sequence of instructions without any jumps, except as last instruction, or jump targets within this sequence.

behavior, results in a very pessimistic WCET bound. Plenty of effort has gone into research to integrate the instruction cache into the timing analysis of tasks [2, 28, 16], the cache's influence on task preemption [38, 11], and integration of the cache analysis with the pipeline analysis [27]. Heckmann et. al described the influence of different cache architectures on WCET analysis [29]. Suggestions for future architectures of memory hierarchies are given in [78]. An overview of WCET related research is presented in [53] and [77].

2.3. Data-Flow Analysis

Using data-flow analysis (DFA) to determine the possible receivers for virtual method calls is a well-established technique. The analysis presented in this paper is similar to k-CFA ("k^{th}-order control-flow analysis") [70, 71]. Techniques like the ones described in [4] or [74] are more efficient than this approach, but less precise. The differences stem from the fact that these analyses use larger scopes to compute information—in the extreme case, a global set of receiver types is computed. Differences also come from the fact that some analyses start from a conservative estimate of the call graph, while our approach builds up the call graph from scratch.

Our approach to loop bound analysis is similar to one of the flavors of abstract interpretation presented by Gustafsson in [18], namely the "approximate interpretation with widening/narrowing". However, our approach differs in the way values are widened/narrowed, as it is tailored to the needs of our analysis. The prototype tool developed by Gustafsson does not use the widening/narrowing technique.

In a different (but in its results somewhat similar) approach, Gustafsson et al. use *abstract execution* to detect loop bounds and infeasible paths [19]. With this approach, loop iterations are analyzed individually. On the one hand, this potentially yields very precise results. On the other hand, some of the potential precision must be sacrificed to keep the cost of the analysis feasible.

The approach of Healy et al. [26] computes information that is fairly similar to the information computed by our loop bound analysis. However, the information is only computed for induction variables. Unlike our approach, it also uses summation to handle nested loops and is therefore superior in this regard.

Prantl [49] extends a simple interval analysis with constraint solving to find loop bounds. The strength of that approach is that it naturally handles nested loops, a feature that lacks from usual loop bound analyses. The benchmarks in the paper indicate that the constraint solving approach and our loop bound analysis detect a similar amount of loop bounds.

In [12], a pattern based and a data-flow based approach to find loop bounds are compared. The data-flow based approach only handles constant additions for loop variables. In contrast, our approach handles additions of bounded values and can therefore be considered slightly more powerful. The comparison to the pattern-based approach shows that the data-flow based approach is superior in most cases. Only in a few special cases, the former approach finds a bound where the latter approach cannot.

2.4. WCET Analysis for Java

WCET analysis at the bytecode level was first considered in [8]. It is argued that the well-formed intermediate representation of a program in Java bytecode, which can also be generated from compilers for other languages (e.g. Ada), is well suited for a portable WCET analysis tool. In that paper,

annotations for Java and Ada are presented to guide the WCET analysis at bytecode level. The work is extended to address the machine-dependent low-level timing analysis [5]. 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. The proposed annotation system can be extended to tighten the set of possible receivers of the dynamic method dispatch [31]. To avoid these relatively complex annotations we use a DFA based receiver analysis. An extension of the Java annotation mechanism (since Java 1.5) for WCET related annotations is proposed in [24].

In [52] a portable WCET analysis is proposed. The abstract WCET analysis is performed on the development 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. In [6], an extension of [8] and [5], an approach how the portable WCET information can be embedded in the Java class file is presented. It is suggested that the final WCET calculation is performed by the target JVM. The calculation is performed for each method and the static call tree is traversed by a recursive call of the WCET analyzer. JVM timing models for the portable WCET analysis can be derived by measurements [32]. However, measurements cannot guarantee safe upper bounds of the timing. We perform static analysis at the processor's microcode level to derive the JVM timing model.

2.5. WCET Analysis for JOP

Strong indications that JOP is a *WCET friendly* design are other real-time analysis projects that target JOP. The first IPET based WCET analysis tool that includes the timing model of JOP is presented in [68]. A simplified version of the method cache, the two block cache, is analyzed for invocations in inner loops. Trevor Harmon developed a tree-based WCET analyzer for interactive back-annotation of WCET estimates into the program source [23, 22]. The tool is extended to integrate a simplified version of JOP's method cache [25]. A project closely related to our model checking approach is presented in [9]. Model checking is used to analyze the timing and scheduling properties of a complete application within a single model. However, even with a simple example, consisting of two periodic and two sporadic tasks, this approach leads to a very long analysis time. In contrast to that approach, our opinion is that a combined WCET analysis and schedulability analysis is infeasible for non-trivial applications. Therefore, we stay in the WCET analysis domain, and consider the well established schedulability analysis as an extra step.

Compared to those three tools, which also target JOP, our presented WCET tool is enhanced with: (a) integration of data-flow analysis for loop bounds and receiver types; (b) analysis of bytecodes that are implemented in Java; (c) a tighter IPET based method cache analysis; and (d) an evaluation of model checking for exact modeling of the method cache.

3. A Time-predictable Java Processor

JOP [62, 64], the Java Optimized Processor, is designed to minimize the *worst-case execution time* instead of the *average case execution time*. The processor architecture is designed to be WCET analyzable. Features such as the real-time stack cache and method cache, provide good performance

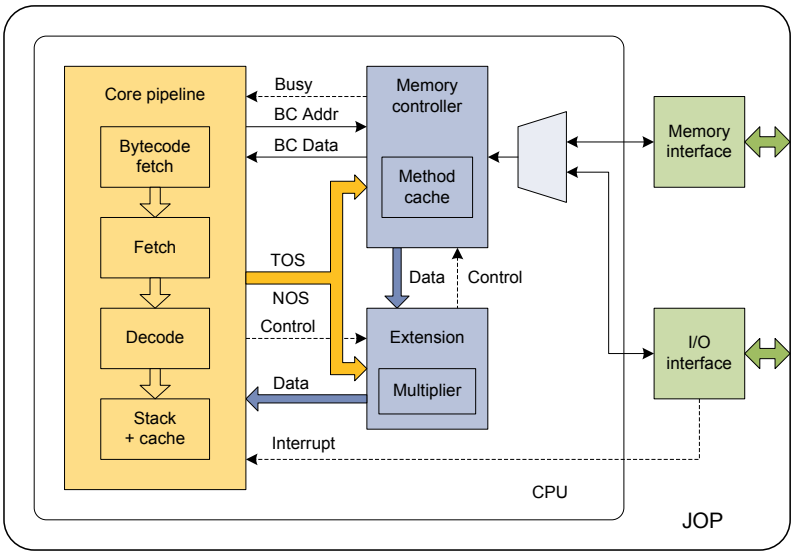


Figure 1. Block diagram of the Java processor JOP

and are still analyzable. The execution time for Java bytecodes can be exactly predicted in terms of the number of clock cycles.

3.1. 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 instructions in hardware. The difference between the JVM and JOP is best described as follows:

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

Figure 1 shows JOP's major functional units. A typical configuration of JOP contains the processor core, a memory controller, a memory interface and a number of I/O devices. Main memory and I/O devices are connected via the memory controller. 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. Therefore, some load time of the method cache can be hidden by microcode instruction for invoke and return.

3.2. 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 the stack spills or fills. 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 [60]. No data dependencies, e.g., a load/use stall, need to be analyzed for this simple pipeline. The short pipeline results in short branch delays. Therefore, a hard-to-analyze branch prediction logic can be avoided. The execution stage, labeled *Stack + cache* in Figure 1, also contains the stack cache.

In [63], we showed that no processor resources are shared across bytecode boundaries. This means that there are no pipeline dependencies between two bytecodes that could generate an unbounded timing effect. The processor is designed to avoid any timing anomalies as found in standard microprocessors [42]. We do not need to model the pipeline in the low-level WCET analysis.

3.3. Caches

Standard cache organizations are difficult to predict for WCET analysis. Two time-predictable caches have been proposed for JOP: a *stack cache* as a substitution for the data cache and a *method cache* to cache the instructions.

3.3.1. 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 invoke and return instructions.

Keeping the stack in the cache for each thread results in faster invokes and returns, but limits the maximum stack height for each thread. The local stack is now part of the thread's context and has to be saved and restored on a thread switch. The additional time has to be added to the context switch time. The maximum height of the stack is bounded if recursions are bounded. The maximum height can be derived by analyzing the call graph for each thread and results in a known maximum context switch time.

3.3.2. Method Cache

Typical Java programs consist of short methods (see [62]). Furthermore, 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 instruction 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. Other instructions are guaranteed cache hits. The fully loaded method and relative addressing inside a method also result in a simpler cache design. A fast tag memory or address translation is not necessary. As the cache stores whole methods, it is named *method cache* [59].

Methods that are larger than the available method cache need a special treatment. In the Java library we found only class initializer methods that are too large. The class initializers usually contain straight line code and are executed once at application start. Therefore, those methods are not cached at all. The class initializer methods are not part of the mission phase and need not be analyzed. Large methods in the application code need to be split into smaller methods. This code transformation is inverse to method inlining and can be performed at source or bytecode level.

The main difference between the method cache and a conventional cache is that the blocks for a method are all loaded at once, and need to be consecutive. A method loaded over the cache end to the cache start is considered contiguous as the cache is addressed with a modulo counting program counter. The constraint that a method has to fill successive blocks in the cache restricts the possible replacement policies. Either a stack based or a FIFO based replacement is feasible. LRU replacement is, although better analyzable [58], unfortunately not possible. The disadvantage of stack based replacement is that two methods in a loop would continually replace each other. Therefore, we have chosen to use a FIFO based replacement policy.

3.4. Comparison with picoJava

We contrast the time-predictable JOP design with picoJava [72, 73], a Java processor designed for average case performance. Simple bytecodes are directly supported by the processor. Most of them execute in a single cycle. More complex bytecodes trap to a software routine. However, the invocation time of the trap depends on the cache state and is between 6 cycles in the best case and 426 cycles in the worst case – a factor in the order of two magnitudes. Some of the trapped instructions (e.g., invokevirtual) can be replaced at runtime by a *quick* version (e.g., invokevirtual_quick). This replacement results in different execution times for the first execution of some code and following executions.

To speedup sequences of stack operations picoJava can fold several instructions into a RISC style three register operation: e.g., the sequence: load, load, add, store. This feature compensates for the inefficiency of a stack machine. However, the folding unit depends on a 16 byte instruction buffer with all the resulting unbounded timing effects of a prefetch queue.

picoJava implements a 64 word stack buffer as discrete registers. Spill and fill of that stack buffer is performed in background by the hardware. Therefore, the stack buffer closely interacts with the data cache. The interference between the folding unit, the instruction buffer, the instruction cache, the stack buffer, the data cache, and the memory interface is complex to model for WCET analysis.

JOP does not contain any pipeline dependencies, which simplifies WCET analysis. Comparing JOP to the advanced architecture of picoJava gives an idea of the performance impact of a simpler pipeline. JOP is between 15% and 30% slower than picoJava on average case measurements [51]. We consider this performance difference a small price that needs to be payed for a time-predictable architecture where tight WCET bounds can be derived.

4. WCET Analysis

In hard real-time systems, the estimation of the WCET is essential. WCET analysis is in general an undecidable problem. Program restrictions, as given in [55], make this problem decidable. Loops and recursion depths need to be bounded. The presented tool uses data-flow analysis to automatically extract (simple) loop bounds. For loop bounds that cannot be automatically determined, source code annotations are supported. Without support for tail-recursive calls, the associated runtime cost suggests not to use recursion in embedded systems with restricted memory. Therefore, we do not support recursion in the WCET tool at the moment.

Another difficulty of high-level WCET analysis are function pointers. The targets of C based function pointers are hard to predict. The more controlled form of dynamic dispatch in object-oriented languages is easier to handle. With class hierarchy analysis all possible receiver methods for an invocation can be found. Data-flow analysis is used to further restrict the set of possible methods. The remaining set of methods is modeled as alternatives in the CFG.

As the full application has to be available for the WCET analysis, we disallow dynamic class loading. For embedded real-time systems this is not a severe restriction. There is more information available in Java class files than in compiled C/C++ executables. All links are symbolic and it is possible to reconstruct the class hierarchy from the class files.

Java bytecode generation has to follow stringent rules [41] 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.[§] Detection of basic blocks in Java bytecode and construction of the CFG is thus straight forward.

4.1. IPET Based WCET Analysis

The high-level WCET analysis, presented in this section, is based on the standard IPET approach [56, 39].

4.1.1. ILP Formulation

The calculation of the WCET is transformed to an ILP problem [56]. 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 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 [3]. Furthermore, Sun's Java compilers version 1.4.2 and later compile finally blocks without subroutines.

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:

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

The frequencies f_i are the integer variables calculated by solving 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. The termination node T has only incoming edges with the sum of the frequencies also set to 1; all return statements of the top-level method are connected to the node T. This means that the program is executed once and can only terminate once through T.

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

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

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 [40], 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:

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

In Section 5, we will show how the cache constraints for the method cache can be integrated into the analysis. An example of the WCET analysis flow with the resulting ILP equations can be found in [68].

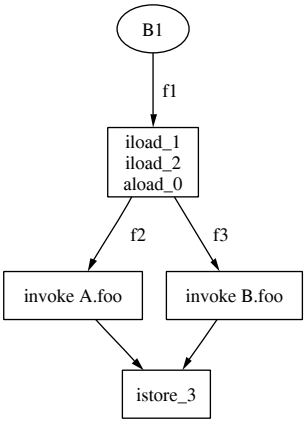


Figure 2. Split of the basic block for possible instance methods

4.1.2. Dynamic Method Dispatch

Dynamic runtime-dispatching of inherited or overridden instance methods is a key feature of object-oriented programming. Therefore, we allow dynamic methods as a *controlled* form of function pointers. The invoked method, selected dynamically on the actual object type, is called a receiver method in object-oriented languages as the object *receives* a message. The full class hierarchy can be extracted from the class files of the application. From the class hierarchy, all possible receiver methods for an invocation can be extracted.

When all possible receivers are included as alternatives, the resulting WCET bound can be very pessimistic. Therefore, we tighten this set by a flow-sensitive receiver analysis (see Section 6). All remaining possible receivers are included as alternatives in the ILP constraints.

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. The example for two classes A and B that are part of the same hierarchy is shown in Figure 2. Following the standard rules for the incoming and outgoing edges the resulting ILP constraint for this example is:

$$f_1 = f_2 + f_3$$

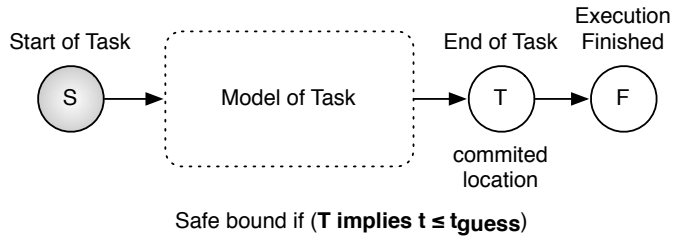


Figure 3. Calculating the WCET bound using UPPAAL

4.2. Model Checking Based WCET Analysis

While the IPET method obtains an implicit representation of the worst-case path by solving a system of flow equations, model checking explores the state space of the program, searching for the worst-case path. A state comprises the current state of the program, parts of the execution history such as the cache state, and constraints on the possible values of real-time clocks.

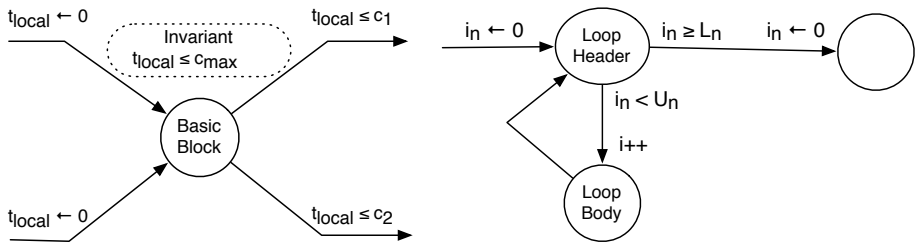
Even though model checking is significantly more expensive than IPET, it offers new possibilities, such as using accurate models of pipelines or caches. Experimental results using the UPPAAL model checker indicate that model checking is fast enough for typical tasks in embedded applications, though large loop bounds or many global variables lead to long analysis times. To cope with more complex applications, model checking needs to be restricted to more important parts of the program and combined with the other techniques such as the IPET approach. Similar to IPET, the model checking analysis requires all loops to be bounded. To simplify the translation, we also omitted support for recursive method invocations.

UPPAAL is a model checker based on networks of timed automata, supporting *bounded integer variables* and synchronization using *channels* [7]. We have implemented a translation of Java programs to UPPAAL models, and use the model checker to determine a safe WCET bound. Our initial attempt was based on ideas from [46, 9] and has been subsequently refined, adding progress measures [37] to reduce the search time, and cache simulations.

4.2.1. General Strategy

An UPPAAL model comprises global clocks, synchronization channels and bounded integer variables, and a set of *processes*. Each process is instantiated from a *template* and may have its own set of local clocks and variables. We start by declaring a global clock t, which represents the total time passed so far, and build processes simulating the behavior of the program, one for each method. Additionally, we add a clock representing the local time passed at some instruction, t_{local}.

There is one location S, which is the entry point of the program, and one location T, which corresponds to the program's exit. When execution has finished, the system takes the transition from T to the final state F (Figure 3). If we want to check whether t_{guess} is a safe WCET bound, we ask the model checker to verify that for all states which are at location T, $t \leq t_{guess}$ holds. If this property is



(a) Modeling basic blocks (b) Modeling loops

Figure 4. Modeling CFGs as timed automata

satisfied, t_{guess} is a safe WCET bound; otherwise we have to assume it is not. Starting with a known upper bound, we perform a binary search to find a tighter WCET bound. Note that if the model checker kept track of the maximum value of t encountered during state exploration, it would not be necessary to perform a binary search.

4.2.2. Translation of Control Flow Graphs

Given the CFG of a Java method m_i, we build a process M_i simulating the behavior of that method by adding *locations* representing the CFG's nodes and *transitions* representing the flow of control. The initial location $M_i.S$ corresponds to the entry node of the CFG, and the location $M_i.T$ to its exit node.

To model the timing of basic blocks, we reset t_{local} at the incoming edges of a basic block. If the execution of the basic block takes at most c_{max} cycles, we add the invariant $t_{local} \leq c_{max}$ to the corresponding location. On architectures where the maximum number of cycles depends on the edge i taken, additional guards of the form $t_{local} \leq c_i$ are added to the outgoing edges of the basic block (Figure 4(a)).

4.2.3. Modeling Loops

It would be possible to eliminate bounded loops by unrolling them in a preprocessing step, but it is more efficient to rely on bounded integer variables. Assume it is known that the body of loop n is executed at least L_n and at most U_n times. We declare a local bounded integer variable i_n representing the loop counter, ranging from 0 to U_n. The loop counter is initialized to 0 when the loop is entered. If an edge implies that the loop header will be executed one more time, a guard $i_n < U_n$ and an update $i_n \leftarrow i_n + 1$ is added to the corresponding transition. If an edge leaves the loop, we add a guard $i_n \geq L_n$ and an update $i_n \leftarrow 0$ to the transition (Figure 4(b)).

It might be beneficial to set $L_n = U_n$, but this is only correct in the absence of timing anomalies, and therefore in general unsound in the presence of FIFO caches. Assuming an empty cache for FIFO replacement policy is not the worst case – therefore, we consider FIFO replacement as a form of timing

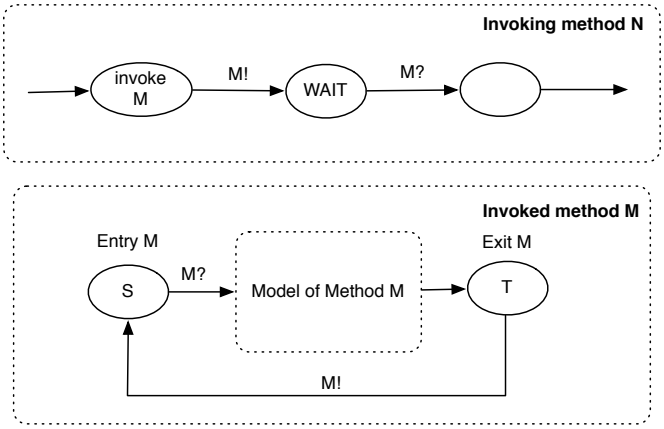


Figure 5. Translation of method invocations

anomaly. In principle, every control flow representable using bounded integer variables can be modeled using UPPAAL, though we only implemented relatively simple loop bounds in the current version of the WCA tool.

4.2.4. Method Invocations

We instantiate one process M_i for each reachable method m_i. To model method invocations, we synchronize the processes using synchronization channels. To allow the method to be invoked several times, a transition from $M_i.T$ to $M_i.S$ is added to all methods. When a method m_i is invoked, the invoke transition synchronizes with the outgoing transition of $M_i.S$ on the invoked method's channel. When returning from method m_i, the transition from $M_i.T$ to $M_i.S$ synchronizes with the corresponding return transition in the calling method. This translation assumes that there are no recursive calls (see Figure 5).

4.3. 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 documented. In [63], 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 further measurements are necessary. We have shown that there are no dependencies, neither through pipeline effects nor shared processor

Table I. Execution time of simple bytecodes in cycles

Instruction	Cycles	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

resources, between individual bytecodes. The detailed bytecode instruction timing can be found in [62] and [65].¶

4.3.1. Basic Bytecodes

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 on-chip, 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 I 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.

4.3.2. 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). With a memory with r_{ws} wait states, the execution time for, e.g., getfield is

$$t_{\mathit{getfield}} = 11 + 2r_{ws}.$$

A memory read in JOP microcode is split into two phases: (1) start the read transaction and (2) read the result. These two phases can be considered an explicit version of the load delay slot in an early version of the MIPS CPU. The MIPS delay slot needs to be scheduled by the compiler, whereas the microcode in JOP is optimized by hand. To avoid timing dependencies within the memory subsystem over bytecode boundaries, memory store instructions are also split into a start write and wait for

¶The latest revision of the timing information is available at `http://www.jopdesign.com/`

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 ldc2_w in clock cycles:

$$t_{ldc2\_w} = 17 + \max(r_{ws} - 2, 0) + \max(r_{ws} - 1, 0)$$

Thus, for a memory with two cycles access time ($r_{ws} = 1$), as we use it for a 100 MHz version of JOP with a 15 ns 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 f_{ws} for a single word cache fill is:

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

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

$$l = \begin{cases} 6 + (n+1)(1+f_{ws}) & : \text{ cache miss} \\ 4 & : \text{ cache hit} \end{cases}$$

where n is the size 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:

$$t_{invokestatic} = 74 + r_{ws} + \max(r_{ws} - 3, 0) + \max(r_{ws} - 2, 0) + \max(l - 37, 0)$$

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

4.3.3. Bytecodes in Java

More complex bytecodes can be implemented in Java on JOP. This feature is used, for example, to implement floating point instructions in software. If such a software bytecode is executed, a static method from a JVM internal class gets invoked. For the WCET analysis this bytecode is substituted by an invoke instruction to this method. The influence on the cache (the bytecode execution results in a method load) can be analyzed in the same way as for *ordinary* static methods (see Section 5).

4.3.4. Native Methods

Most of the JVM internal functionality in JOP, such as input, output, and thread scheduling, is implemented in Java; i.e., in software. However, Java and the JVM do not allow direct access to memory, peripheral devices, or processor registers. For this low-level access to system resources, we need *native* methods. 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 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.

4.3.5. *Calculation of Bytecode Timings*

To avoid the recalculation of the bytecode timings every time the microcode changes, we implemented an analysis tool to derive the bytecode timings from microcode assembler. This is less error prone than a manual calculation, and allows to adapt the microcode to the needs of a particular platform without lengthy recalculations of low-level timing properties.

The number of cycles needed to execute a bytecode depends on the set of possibly executed microcode instruction sequences, and the latency of memory accesses. For bytecodes that execute the stcbrd microcode (which starts the method cache fill on a miss), the time needed for a memory access not only depends on the read and write delay, but also on the size of the accessed method and whether the method cache access is a hit or miss.

The tool first extracts the set of microcode instruction sequences for a bytecode, and then calculates a timing formula for each instruction sequence. The tool can handle all bytecodes automatically, expect those, where the *timing of the microcode part* depends on values not known at compile time (e.g., int2ext, which is used to replace the stack on a context switch).

First, the set of microcode instruction sequences for a bytecode is extracted from the corresponding assembler code. To this end, the assembler label for the bytecode is identified, and the assembler code starting at this label is executed in an abstract interpreter. The interpreter keeps track of known constant values, so it is possible to deal with bounded loops and distinguish between memory access to main memory (with wait states) and on-chip I/O devices. If the value on top of the stack is unknown during interpretation of a conditional jump, the interpreter stores a continuation and at a later point generates a microcode path for the other alternative. As soon as the length of some microcode path exceeds a user-defined threshold, the analysis gives up. In this rare case (currently only two bytecode implementations are affected), the bytecode has to be analyzed manually.

Next, we generate a timing formula, as presented in Section 4.3.2, for each microcode path. Simple microcodes, which do not access the memory subsystem, contribute one cycle to the bytecode timing. Microcodes which access the memory execute in parallel. Two consecutive wait microcodes block until the memory access is finished. In between, microcodes which do no access the memory may be issued. The total number of cycles for such a sequence thus depends on the microcodes issued during the memory access, and the minimum number of cycles needed for executing the memory accesses.

For example, the memory access for the stmwd (memory write) microcode needs at least $2 + w$ cycles. Now consider the microcode sequence

```
stmwd
ldm moncnt
 ldi  1
add
wait
wait
```

Together with the starting first and synchronizing last microcode, the sequence thus takes at least $4+w$ cycles. As a total of 6 microcodes are issued, the sequence also takes at least 6 cycles. Therefore, the time needed to execute the above microcode sequence is $\max(4 + w, 6)$.

Additionally, several invariants are verified during microcode analysis. For instance, we check that at most one memory-accessing microcode is executed at a time, that the top of stack is not be modified

prior to a jump, and that a certain number of cycles pass between the start of a multiplication, and reading the result.

Taking the maximum of the timing formulas extracted for each microcode path, a parametrized formula for the bytecode timing is obtained. When the configuration of the memory, the size of the invoked method and the cache access classification is known during WCET analysis, this formula finally provides an integer bounding the actual number of cycles needed to execute the bytecode.

4.4. Implementation

The WCET analyzer (WCA) is an open source Java program and implements the IPET based and model checking based WCET analysis. In contrast to the former description, WCA associates the execution cost to the edges instead of the vertices. For modeling the execution time of JOP those two approaches are equivalent. However, WCA is prepared for other architectures with different execution times of a branch taken or not taken.

To access the class files, we use the Byte Code Engineering Library (BCEL) [13]. BCEL allows inspection and manipulation of class files and the bytecodes of the methods. The WCA extracts the basic blocks from the methods and builds the CFG. Within the CFG, the WCA detects both the loops and the loop header. 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 the WCA by directly invoking it via the Java library binding.

In addition to the usual approach modeling the application as one big ILP, we also support a recursive top-down analysis, which first analyzes referenced methods separately and then solves the inter-procedural ILP model. This is a fast, simple alternative to the global model, and allows us to invoke the more expensive model checker for smaller, important parts of the application. Furthermore, the recursive analysis is able to support incremental updates, and could be used by a bytecode optimizer or an interactive environment.

After completing the WCET analysis, WCA creates detailed reports to provide feedback to the user and annotate the source code as far as possible. All reports are formatted as HTML pages. Each individual method is listed with basic blocks, execution time of bytecodes, and cache miss times. The WCA also generates a graphical representation of the CFG for each method and for the whole program. Furthermore, the source is annotated with execution times and the WCET path is marked, as shown in Figure 6. For this purpose, the solution of the ILP is analyzed, first annotating the nodes of the CFG with execution costs, and then mapping the costs back to the source code.

5. Cache Analysis

From the common usage and the properties of the Java language — usually small methods and relative branches — we derived the novel idea of a *method cache* [59], 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 cache misses only occur at two instructions (*invoke* and *return*), the instruction cache can be ignored on all other instructions. The time to load a method depends on the method size and the properties (latency and

```
34          |         public static int measure(boolean b, int val) {
35          |
36          |                 int i, j;
37          |
38  [159]   |                 for (i=0; i<10; ++i) {
39  [50]    |                         if (b) {
40  [1160]  |                                 for (j=0; j<3; ++j) {
41          |                                         val *= val;
42          |                                 }
43          |                         } else {
44          |                                 for (j=0; j<4; ++j) {
45          |                                         val += val;
46          |                                 }
47          |                         }
48          |                 }
49  [24]    |                 return val;
50          |         }
```

Figure 6. WCET path highlighting in the source code

bandwidth) of the memory interface. On an invoke, the size of the invoked method is used, and on a return, the method size of the caller.

5.1. IPET Based Cache Analysis

To integrate the method cache into the WCET analysis, cache accesses are modeled inserting extra vertices and edges in the timing graph. The frequencies of cache access edges are then restricted to reflect the worst-case behavior of the cache.

As the cache hits or misses can only happen at method invocation, or return from a method, we model cache access times by inserting extra vertices before and after nodes which trigger a method invocation.

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 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, to ensure that one block contains only the invoke instruction. Cache accesses on invoke and return are modeled by inserting an additional vertex representing a cache miss, and two edges for a cache miss and hit, respectively.

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. This approach simplifies the analysis as both methods, the caller and the called, with their respective sizes, are known at the invoke instruction.

Figure 8 shows the resulting graph after the split of block B4 and the inserted vertices for the cache misses. The miss penalty is handled in the same way as execution time of basic blocks for the ILP

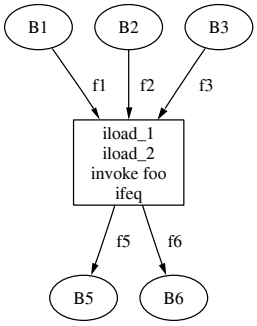


Figure 7. Basic block with an invoke instruction

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 and miss frequencies f_{ih} and f_{im} and the return hit and miss frequencies f_{rh} and f_{rm}.

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 [42]. As a next step, we have to formulate the relation between the hit and the miss frequency.

The method cache is divided in several blocks similar to cache lines in a conventional cache. However, a single method has to be loaded in a contiguous region of the cache. The effective replacement policy of the method cache is FIFO. A LRU replacement would be preferable, as it provides a slightly better caching behavior and is easier to analyze. However, the constraint on the method cache that a method has to span several contiguous blocks makes the implementation of the LRU replacement strategy difficult and costly in terms of hardware resources.

Due to the FIFO replacement, one can construct examples where a method in a loop can be classified as single miss, but that miss does not need to happen at the first iteration. As, furthermore, a long cache access history is needed to classify a cache access as hit or as miss, standard classification techniques perform poorly on FIFO caches [58].

Therefore, instead of classifying one particular cache access, we bound the number of cache misses during the execution of a method or loop. The approach is based on the following fact for N block FIFO caches: If it is known that during the execution of a code region at most N distinct cache blocks are accessed, each accessed block will be loaded *at most once*. For example, suppose all methods invoked during the execution of a loop, together with the method containing the loop, fit into the cache. Then for every method possibly accessed during the execution, at most one cache access will be a cache miss.

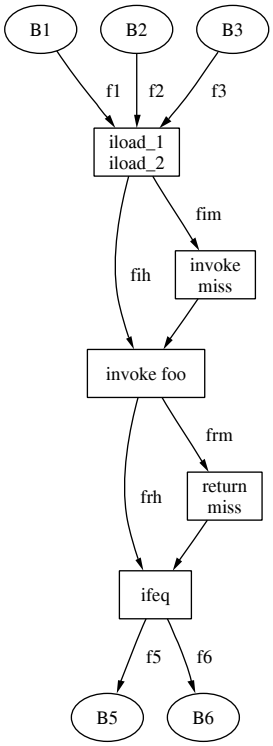


Figure 8. Split of the basic block and cache miss blocks

To use this fact, we need to find a bound on the maximum number of *distinct* cache blocks $\mathrm{maxBlocks}(m)$, which might be possibly accessed during the execution of any method m. The quality of the cache analysis hence depends on the heuristic used to check whether at most N distinct cache blocks are accessed. We found that simply considering all methods possibly invoked works well enough, but better results may be obtained using a more sophisticated check.

The cache analysis proceeds as follows: The methods of the analyzed task are analyzed, starting from the root method. When analyzing a method, we check for each invoke(m) block, whether $\mathrm{maxBlocks}(m) \leq N$. If this is not the case, we continue inspecting the CFG of m.

Otherwise, the analysis recursively duplicates the CFG of the invoked method m and of all methods possibly invoked during its execution. This is necessary, as the cache miss frequency constraints should only apply to those method invocations where m is a parent in the call stack.

Then, for each method n possibly invoked during the execution of m, we identify all edges $f_n \in E_{miss(n)}$ representing a cache miss of n. Let e_m denote the the execution frequency of method m. To reflect the fact that n will be loaded at most once during one execution of m, we add the following linear constraint for n:

Listing 1. FIFO Cache Analysis

```
m() {
    a();  /* f1 (invoke a), f2 (return to m) */
    b();  /* f3 (invoke b), f4 (return to m) */
}
a() {
    b();  /* f5 (invoke b), f6 (return to a) */
    c();  /* f7 (invoke c), f8 (return to a) */
}
b() {
    c();  /* f9 (invoke c), f10 (return to b) */
}
c() {
}
```

$$\sum_{f_n} \in E_{miss(n)} f_n \leq e_m$$

An example, shown in Listing 1, illustrates the cache analysis. Assume that methods a, b, and c together fit into the cache, but not b, c, and m. The cache miss frequency variables for the methods in the example are given in the comment, e.g., f_1 for a miss on invoking method a and f_2 for a miss when method a returns. During the execution of a, as all three methods fit into the cache, each of the methods a, b and c will be missed at most once. Similarly, during the execution of b called from m, both b and c will be loaded at most once.

The cache analysis starts by inspecting m, and identifies that $\mathrm{maxBlocks}(a) \leq N$. Therefore, a, b and c are duplicated, resulting in methods a', b' and c', and cache miss frequencies f'_i. The following ILP constraints are generated:

$$\begin{aligned} f_1 + f'_6 + f'_8 &\leq e'_a \\ f'_5 + f'_{10} &\leq e'_a \\ f'_7 + f'_9 &\leq e'_a \end{aligned}$$

Due to these constraints, e.g., either f'_5 or f'_7 might be executed once per execution of a, but not both of them.

The analysis continues to inspect m, and finds that $\mathrm{maxBlocks}(b) \leq N$. The renaming results in methods b'' and c'', and cache miss frequencies f''_i. Finally, these ILP constraints are added:

$$\begin{aligned} f_3 + f''_{10} &\leq e''_b \\ f''_9 &\leq e''_b \end{aligned}$$

The technique presented above is easy to apply and only needs to perform simple static analysis. In contrast to a direct-mapped instruction cache, we only need to consider invoke instructions and do not need to know the actual bytecode address of any instruction, thus increasing the portability of the analysis.

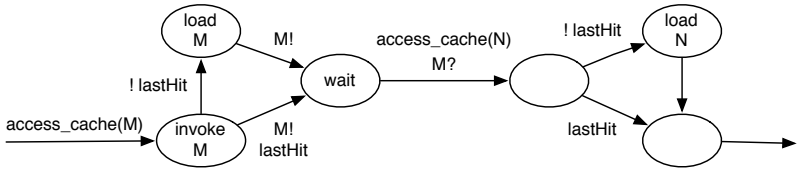


Figure 9. Modeling the method cache accesses

5.2. Method Cache Simulation

Using timed automata, it is possible to directly include the cache state into the timing model. It is most important, however, to keep the number of different cache states low, to limit space and time needed for the WCET calculation. JOP's method cache is especially well suited for model checking, as the number of blocks and consequently the number of different cache states are small.

To include the method cache in the UPPAAL model, we introduce an array of global, bounded integer variables, representing the blocks of the cache. It is assumed that the cache initially only contains the main method and is otherwise empty. As this is not a safe approximation in general, we have to ensure that the first access to some method is actually a cache miss, for example by inserting a cache flush at the beginning of the main method.

We insert two additional locations right before and after the invoke location, modeling the time spent for loading the invoked method and the invoking method at a return, respectively. The UPPAAL function access_cache updates the global cache state and sets the variable lastHit to either true or false (see Figure 9 and Listing 2).

6. Data Flow Analysis

WCET analysis has to reason about the behavior of programs, especially with regard to loop bounds. The information about the constraints for the program flow are also referred to as flow facts. Many approaches use manual annotations to communicate flow facts to the analysis tool. A survey on annotation languages can be found in [35]. Manual annotation of flow facts is not an ideal solution. The respective information is specified in two places: The program code and the annotation itself. Inevitably, this can lead to inconsistencies, especially if the logic of a program is updated. It is not trivial to prove that an annotation is correct (the work of [33] aims to achieve this by using the Java Modeling Language). In some cases, it would also be necessary to annotate library code (e. g., String.equals()), which is not always possible.

Data-flow analysis (DFA) can solve many of these problems and remove the need for a number of annotations. Context sensitive DFA can also naturally model context-sensitive behavior; modeling such behavior through manual annotations is cumbersome and requires powerful annotation languages. However, in general, DFA cannot find all loop bounds. Some knowledge of loop bounds resides outside the program itself, e.g., input data dependent loops or when a loop depends on the behavior of an I/O

Listing 2. FIFO cache simulation

```
const int NUM_BLOCKS[NUM_METHODS] = { /* number of blocks per method */};
const int EMPTY_TAG = NUM_METHODS;
int [0,NUM_METHODS] cache[NUM_BLOCKS] = { EMPTY_TAG, ROOT_METHOD, EMPTY_TAG, ... };
bool lastHit ;

void access_cache(int mId) {
    int i = 0;
    int sz = NUM_BLOCKS[mId];
    lastHit = false;
    for(i = 0; i < NUM_BLOCKS; i++) {
        if (cache[i] == mId) {
            lastHit = true;
            return;
        }
    }
    for(i = NUM_BLOCKS - 1; i >= sz; i--) {
        cache[i] = cache[i-sz];
    }
    for(i = 0; i < sz-1; i++) {
        cache[i] = EMPTY_TAG;
    }
    cache[i] = mId;
}
```

device. Some knowledge is lost due to the abstractions an analysis makes (e.g., modeling the possible values of a variable as an interval); such abstractions are however required to keep the computational effort of the analysis feasible. Still, DFA can minimize the need for annotations and therefore also the effort required to correctly annotate a program.

Furthermore, a WCET analysis tool for object oriented languages requires knowledge about the receivers of invoked methods in order to minimize the pessimism for virtual method calls. In this section, a data-flow analysis framework is presented, which was developed to aid WCET analysis and works directly on Java bytecode. Currently, a receiver type analysis and a loop bound analysis are implemented.

6.1. The Analysis Framework

We implemented an analysis framework that operates directly on Java bytecodes. Unlike machine code, Java bytecode contains symbolic information about types, field names etc. As JOP executes Java bytecode with only minor modifications,[‖] an analysis at this level is able to model the program behavior

[‖]Most notably, the binaries for JOP are prelinked and symbolic information is stripped. However, this does not change the semantics of the program.

on JOP accurately. Having symbolic information at hand also allows easy interfacing with the WCET analysis tool. The framework builds on other classes within the tool chain for JOP, e. g., to build the transitive hull of classes. We chose not to use an analysis framework like PAG [1] or soot [57], because the overheads for switching from the existing infrastructure to such a framework did not seem to be justified.

The infrastructure and the analysis kernel are strongly influenced by the algorithms described in [44]. The analyses themselves resemble traditional bytecode interpreters, but operate on abstract states, e.g., intervals of integer values. All analyses implement a common interface, which comprises methods to create initial information and to compare, join and transfer the information. The analysis kernel itself is a classic worklist algorithm.

6.1.1. Bootstrapping

Although being the starting point of program execution, the main() method is not an appropriate starting point for the analysis. A JVM initializes various data structures before the main() method is invoked. In the case of JOP, the class initializers are also executed before the start of the program. These initializations have to be reflected in the analysis. In the tool described in this paper, a dummy <prologue> method is automatically generated. It invokes the methods for the JVM internal initializations, the class initializer methods, and the main() method. The <prologue> method can be regarded as counterpart of JOP's Startup.boot() method.

6.1.2. Method Invocation

Virtual method calls are obviously a key feature of an object oriented language like Java. Therefore, the method to be invoked is not known a priori at the call site. This in turn means that the invocation of a method cannot be hardcoded into the program flow representation of a DFA. Our approach to this issue is straight forward: The transfer function handles method invocations by recursively analyzing them. For a method invocation, the transfer function therefore has to: (a) set up the information to be passed on, (b) decide which methods may be invoked, (c) analyze the possibly invoked methods, and (d) join the returned information.

To reduce the number of possible receivers, the first analysis to be run is the receiver type analysis, as described in the following section. It determines on the fly which virtual methods may be invoked and then recursively analyzes the appropriate methods. The call graph is therefore built up dynamically in this analysis. Analyses that follow the receiver type analysis can rely on the already computed call graph.

A draw-back of this solution is that exceptions cannot be handled easily, because an exception may trigger a transfer of control across methods. The proper handling of exceptions is therefore still an open issue for our framework.

6.2. Receiver Types

The receiver type determines which version of a method is actually invoked. As this depends on the type of the object, the analysis propagates the possible types for all fields and local variables through the potential execution paths. The bytecode new, for example, creates a mapping from the top-of-

Listing 3. Receiver type analysis example in Java source

```
...
Baz.baz = new Bar();
...
```

Listing 4. Receiver type analysis example in bytecode representation

```
...
13: new class Bar();
16: dup;
17: invokespecial Method Bar.<init>:()V;
20: putstatic  Field  Baz.baz:LBar;
...
```

stack element to the type of the created object. A putfield instruction takes the type of the appropriate stack entry, adds the field name and creates a mapping from this field location to the type of the top stack entry. When allocating arrays, the allocation site is added to the type information. This helps to distinguish arrays that are instantiated in different places and thus makes the computed information more precise.

The receiver type analysis maps each $Location$ to a set of $Types$. A $Location$ represents a stack entry or a location on the heap. A stack entry is modeled as an integer value $s \in \mathbb{N}$, "pointing" to the according stack entry. A heap location consists of a $Type$ and a field name; in case of an array type, all array fields are merged to a single field. A $Type$ is a string that contains the fully qualified class name. For arrays and strings it also contains the position at which the instance of this $Type$ was created (or loaded from the constant pool). Adding the allocation site allows to distinguish array and constant string instances in later analyses. It would also be possible to add the allocation site for regular classes, but the analysis then quickly becomes too expensive in terms of both memory consumption and computation time.

Consider the code fragments in Listings 3 and 4; Listing 4 shows the bytecode created by the Java code in Listing 3. The bytecode at position 13 creates a new mapping for the top stack entry, e. g., stack slot 3, resulting in the mapping $\{\ldots,\ stack[3] \rightarrow \{\texttt{Bar}\},\ \ldots\}$. The subsequent dup bytecode yields the following mapping: $\{\ldots,\ stack[3] \rightarrow \{\texttt{Bar}\},\ stack[4] \rightarrow \{\texttt{Bar}\},\ \ldots\}$. The invocation of the constructor pops off the top element and therefore results in $\{\ldots,\ stack[3] \rightarrow \{\texttt{Bar}\},\ \ldots\}$. Finally, putstatic stores the top stack entry to the static field Baz.baz. The result is the mapping $\{\ldots,\ \texttt{Baz.baz} \rightarrow \{\texttt{Bar}\},\ \ldots\}$. The analysis conceptually does the same as an actual JVM, but uses symbolic locations instead of actual references to objects.

When invoking a virtual method, the analysis looks up all mappings for the appropriate stack entry, and accordingly considers all potentially called methods. Moreover, the analysis records the receiver types for field accesses, such that subsequent analyses can reuse this information.

Our analysis builds up the callgraph step by step, and remembers information for each bytecode. Therefore, it is similar to k-CFA ("k^{th}-order control-flow analysis") [70, 71]. As our current

implementation does not distinguish different call strings, which represent the invocation sequence, it can be classified as 0-CFA. In our experiments (see Section 7), the analysis turned out to be rather complex for some programs. Therefore, we abstained from adding call strings, to keep the cost for the analysis moderate.

The analysis is flow sensitive and computes information for individual bytecodes. Therefore, it is potentially more precise than other analyses, but it also requires more effort than the techniques presented in the study of Bacon [4] or the approach in [74]. Implementing one of these techniques would probably be necessary to handle larger programs.

The WCA does not support recursive method invocations, as their use is discouraged on Java processors due to the high runtime overhead, and they complicate the analysis considerably. Therefore, the static call graph, generated by analyzing the class hierarchy or using the results from the dataflow analysis, has to be acyclic. Using the results of the receiver type dataflow analysis, it is nevertheless possible to support certain uses of the popular delegator pattern (Listing 5). This extends the class of programs for which the WCET can be computed.

6.3. Loop Bounds

To determine the loop bounds in a program, we compute the possible values of integer variables. As it would be infeasible to do this precisely, the analysis operates on intervals of integer values. Integer operations operate on these intervals, pushing a constant c onto the stack results in an interval $[c, c]$. If an input to an integer operation is unknown or if an operation cannot be reasonably defined on intervals, it results in the interval $[-\infty, \infty]$.

To keep the analysis reasonably precise while still being efficient, we also remember the constraints that appear when a conditional jump depends on the value of some variable. The third "ingredient" for the analysis is an interval for the possible increments of a variable. Other operations than incrementing and adding or subtracting known intervals are not supported and result in an increment interval $[-\infty, \infty]$.

We define an $Interval$ as follows:

$$Interval = \{\bot\} \cup \{[l, u] | l \leq u, l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{\infty\}\}$$

We can define the $\sqsubseteq$ relation on two $Interval$ values $int_1 = [l_1, u_1]$ and $int_2 = [l_2, u_2]$ as follows:

$$int_1 \sqsubseteq int_2 \Leftrightarrow l_2 \leq l_1 \wedge u_1 \leq u_2$$

The loop-bound analysis operates on tuples of values; the elements of such a tuple $x = \langle assigned, constrained, increment, source, defscope, softinc, cnt \rangle$ are described in Table II. The comparison operator $\sqsubseteq$ operates only on the $assigned$, $constrained$, $increment$, and $softinc$ elements. The $source$, $defscope$ and cnt elements are auxiliary elements that are irrelevant to the comparison. For two tuples $x_1 = \langle a_1, c_1, i_1, s_1, d_1, o_1, n_1 \rangle$ and $x_2 = \langle a_2, c_2, i_2, s_2, d_2, o_2, n_2 \rangle$, we define $\sqsubseteq$ as pointwise application of the $\sqsubseteq$ relation for the intervals and equality for the $softinc$ flag:

$$x_1 \sqsubseteq x_2 \Leftrightarrow a_1 \sqsubseteq a_2 \wedge c_1 \sqsubseteq c_2 \wedge i_1 \sqsubseteq i_2 \wedge o_2 = o_2$$

Joining two intervals $int_1 = [l_1, u_1]$ and $int_2 = [l_2, u_2]$, is straight forward defined as

$$int_1 \sqcup int_2 := [\min(l_1, l_2), \max(u_1, u_2)]$$

Listing 5. Delegator patterns with receiver analysis

```
interface Reader {
    char getChar();
}

static class DelegatingReader implements Reader {
    private Reader reader;
    public DelegationReader(Reader impl) {
        reader = impl;
    }
    /* DFA analysis can prove that there is no recursion here ,
    as the reader field 's type is always ConstReader */
    public char getChar() {
        return reader.getChar();
    }
}

static class ConstReader implements Reader {
    public char getChar() {
        return x   ;
    }
}

main () {
    Reader r = new DelegationReader(new ConstReader ());
    action(r);
}

void action(Reader r) {
    /* DFA analysis determines loop bound N =100 */
    for (int i = 0; i < 100; i ++) {
        r.getChar();
    }
}
```

Joining two tuples involves constraining and widening of intervals.

After the actual analysis, loop bounds can be computed. For conditional bytecodes, information for the true and the false edge is recorded during the analysis. A valid loop bound can only be computed if the following conditions are met:

- The $assigned$ interval at an edge is finite and non-empty,
- the $increment$ interval is not $\perp$ and does not contain zero, and
- $defscope$ is less than the scope number of the condition.

Table II. Information for loop bound analysis

Name	Type	Description
$assigned$	$Interval$	The $assigned$ interval contains all values a variable may hold. The bottom element $\perp$ represents that no information is available yet. The top element is the interval $[-\infty, \infty]$.
$constrained$	$Interval$	An interval that represents the known constraints for a variable. A condition like if (x < 10) yields the interval $[-\infty, 9]$ for the body of the if-statement. Keeping this interval separate from the $assigned$ interval avoids widening the latter when it is not necessary, i.e., if a constraint is implied by a conditional statement outside a loop.
$increment$	$Interval$	An interval that represents all values by which a variable may be incremented. After adding a known interval to a variable, the $increment$ interval contains at least this interval. For example, adding an interval $[2, 2]$ to a value with an $increment$ interval $[1, 1]$ yields the $increment$ interval $[1, 2]$. Other operations than adding or subtracting a known interval result in the $increment$ interval $[-\infty, \infty]$.
$source$	$Location$	The $source$ location indicates that a value is a copy of some other location. This information is necessary to propagate the constraints generated by a condition back to the actual variable. In Java bytecode, conditions evaluate the top stack entry, not directly a variable.
$defscope$	$\mathbb{N}$	Conditional statements and branch targets are assigned unique scope numbers, in the order as they appear to the analysis. When a variable is assigned a value that is not based on the previous value of the variable, the $defscope$ is set to the most recently assigned scope number. If the $defscope$ of a loop variable is greater than or equal to the loop condition scope, no valid bound can be computed. This catches the case where a loop variable is redefined inside the loop body.
$softinc$	$boolean$	Conditional statements set this flag for the variable they depend on. The transfer and join functions take the flag into account, such that a loop bound can only be computed if the variable is incremented in all paths in a loop body. If $softinc$ is true, the variable has not been incremented on all paths.
cnt	$\mathbb{N}$	The cnt value is incremented every time the $assigned$ interval changes. After reaching a threshold, the $assigned$ interval is widened to $[-\infty, \infty]$; reaching a second threshold widens the $constrained$ interval to $[-\infty, \infty]$. By doing so, it is ensured that the information eventually converges.

With $assigned = [assigned_{min}, assigned_{max}]$ and $increment = [increment_{min}, increment_{max}]$, the bound is computed as

$$bound = \left\lceil \frac{assigned_{max} - assigned_{min} + 1}{\min(|increment_{min}|, |increment_{max}|)} \right\rceil$$

If both edge bounds are valid, the bound for a loop that is controlled by the respective conditional bytecode is the maximum of the two edge bounds. In any other case, no valid loop bound can be computed.

The loop bound analysis is inter-procedural and propagates information between different methods. Furthermore, the analysis is able to compute call-string dependent loop bounds. As it takes the length of arrays and strings into account, it can usually bound loops that iterate through the elements of these data types.

The analysis is agnostic of loops and considers all conditional bytecodes. On the one hand, this approach allows finding value ranges for usual if-statements and can therefore be used to find infeasible paths. On the other hand, the analysis cannot distinguish operations inside a loop from operations outside a loop. Therefore, the analysis cannot bound some loops where the loop variable is modified outside the loop body.

7. Evaluation

In the previous sections, we have described WCET analysis, cache analysis, and DFA analysis. In this section, the tools are evaluated with analysis and measurements of several benchmark programs. The measurement gives us confidence that we have no serious bugs in the analysis and an idea of the pessimism of the analyzed WCET bound. It has to be noted that we actually cannot guarantee to measure the real WCET. If we could measure the WCET, we would not need to perform the WCET analysis at all.

Furthermore, the WCET analysis gives a safe bound, but this bound may be conservative. Due to the abstractions in the analysis, the WCET bound may not be associated with a real feasible execution path. There is no general guarantee that we have knowledge about the worst case path.

The analysis time, which is important for practical use, is evaluated in Section 7.2.3. IPET based WCET analysis with the static cache analysis usually finishes in less than a second for all benchmarks. Data-flow analysis can consume several minutes on more challenging benchmarks.

7.1. Benchmarks

The benchmarks used are shown in Table III, with a short description and the source code size in lines of code (LOC). We created three benchmark groups: small kernel benchmarks, WCET benchmarks provided by the Mälardalen Real-Time Research Center, and three real-world applications.

The benchmark crc calculates a 16-bit CRC for short messages. LineFollower is the code of a simple line-following robot. The SVM benchmark represents an embedded version of the machine learning algorithm called support vector machine (SVM). The SVM is modified minimally to work in the real-time setting [47]. The change is that one data flow dependent path from the initialization phase is bounded prior to deployment. The real-time part of the SVM is only 115 LOC. However, the full code, including non-real time initialization, is 3079 lines. As DFA also considers the initialization code, the size of the initialization code has an impact on the analysis time.

The second group are benchmarks provided by the Mälardalen Real-Time Research Center [43] and ported to Java by Trevor Harmon [25]. As the benchmarks are not very object oriented, the results for the receiver type analysis are not representative and therefore not further evaluated in this section.

The benchmarks Lift and Kfl are real-world examples that are in industrial use. Lift is a simple lift controller, where Kfl is a node in a distributed mast control application. The EjipCmp benchmark is an embedded TCP/IP stack written in Java with an example application consisting of a UDP/IP server and a UDP/IP client, exchanging messages via a loopback network layer driver. The TCP/IP stack and the example applications are optimized for a chip-multiprocessor system (e.g., with non-blocking communication). In our setup, however, we execute all five tasks serially on a uniprocessor and perform

Table III. WCET benchmark examples

Benchmark	Program	Description	LOC
Kernel	crc	CRC calculation for short messages	8
	LineFollower	A simple line follower robot	89
	SVM	Embedded machine learning algorithm	115
Mälardalen	BinarySearch	Binary search program	78
	Bubble	Bubblesort program	63
	Crc	Cyclic redundancy check	154
	ExpInt	Exponential integral function	89
	FDCT	Fast discrete cosine transform	222
	Fibonacci	Simple iterative Fibonacci calculation	37
	InsertionSort	Insertion sort program	60
	JanneComplex	Complex nested loops	72
	MatrixCount	Count numbers in a matrix	85
	MatrixMult	Matrix multiplication	104
	NestedSearch	Search in a multi-dimensional array	487
	QuickSort	Quicksort program	166
	Select	Select smallest element from array	136
	SLE	Simultaneous linear equations	128
Applications	Lift	Lift controller	635
	Kfl	*Kippfahrleitung* application	1366
	EjipCmp	Multicore UDP/IP benchmark	1892

Table IV. JOP benchmark configuration

Memory subsystem property	Value
Read access time	2 cycles
Write access time	3 cycles
Stack cache	1 KB
Instruction cache	4 KB/16 blocks

WCET analysis for the aggregation of the five tasks. The configuration of JOP for the evaluation is listed Table IV.

7.2. WCET Analysis and Measurements

In this section we show the WCET analysis results and compare them with measurements in the target hardware. We also show the results of the DFA based loop bound analysis and report the time it takes to analyze the benchmarks. The WCET results given in this section are based on the IPET based analysis.

Table V. Measured and estimated WCET with result in clock cycles

Program	Measured (cycles)	Estimated (cycles)	Pessimism (ratio)
crc	1449	1513	1.04
LineFollower	2348	2411	1.03
SVM	104880	107009	1.02
BinarySearch	631	636	1.01
Bubble	1262717	1815734	1.44
Crc	191825	383026	2.00
ExpInt	324419	429954	1.33
FDCT	19124	19131	1.00
Fibonacci	1127	1138	1.01
InsertionSort	8927	15410	1.73
JanneComplex	886	6991	7.89
MatrixCount	16420	16423	1.00
MatrixMult	1088497	1088509	1.00
NestedSearch	51777	64687	1.25
QuickSort	9942	115383	11.61
Select	5362	55894	10.42
SLE	20408	50514	2.48
Lift	5446	8466	1.55
Kfl	10411	39555	3.80
EjipCmp	15297	21965	1.44

Difference between IPET and model checking based WCET analysis is only due to cache analysis and is shown in the following section.

7.2.1. Analysis and Measurements

Table V shows the measured execution time and the analyzed WCET. The last column gives an upper bound of the pessimism of the WCET analysis. The WCET analysis result of a method is defined as follows: WCA reports the WCET bound of the executable code within a method including the return statement. The invoke of the to-be-analyzed method is not part of the analysis. With respect to cache analysis the return from the method does not include any miss time – invoke and return miss times are considered as part of the invoke instruction. For the measurement we use a cycle counter and take a time stamp before the first instruction of the method and after the return from the method.

For very simple programs, such as crc, robot, and SVM the pessimism is quite low. The same is true for several of the benchmarks from the Mälardalen benchmark suite. Some are almost cycle accurate. The small overestimation of, e.g., 5 cycles for BinarySearch, results from our measurement methodology and the definition of what the WCET of a single method is. To avoid instrumenting all benchmarks with time stamp code we wrap the benchmarks in a method measure(), vary which

benchmark is invoked in measure(), and analyze this method. Therefore, measure() contains an invoke and the control is returned to measure(). In the case of small benchmarks the method cache still contains measure() on the return from the benchmark method. However, WCA does not include measure() as a possible method for the static method cache analysis (it is performed on a invoke and no invoke of measure() is seen by WCA). Therefore, the return from the benchmark method is classified as miss.

The pessimism of most of the other benchmarks, including the two applications Lift and EjipCmp, are in an acceptable range below a factor of 2. Three benchmarks, JanneComplex, QuickSort, and Select, stand out with a high pessimism. JanneComplex is an artificial benchmark with a complex loop dependency of an inner loop that cannot be handled by our DFA and the annotation results in a conservative estimation. Select also contains nested loops with data dependencies that cannot be expressed with our current annotation syntax. However, it has to be noted that the benchmarks from Mälardalen are designed to challenge WCET tools. We assume that a safety-critical programmer would not use QuickSort as a first choice for time-predictable code.

The pessimism of Kfl is quite high. It 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. If we omit the non-feasible path to the service routine the WCET drops down to 22875 and the overestimation is 2.2.

The Kfl example shows 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 the design to a WCET aware style. In one extreme, this can end up in the single-path programming style [54]. A less radical approach can use some heuristics for a WCET aware programming style. For instance, avoid the service routines in the application main loop.

The results given in Table V are comparable to the results presented in [68]. Most benchmarks now take less cycles than in 2006 as the performance of JOP has been improved (e.g., with hardware support for field and array instructions). The exceptions are two benchmarks, LineFollower and Kfl, which now have a higher WCET. The line-following robot example was rewritten in a more object-oriented way. Finally, the different number for the Kfl resulted from a bug in the WCA that was corrected after the publication in 2006.

7.2.2. Loop Bound Analysis

It is important to detect most of the loop bounds automatically in order to lessen the burden for user. In addition, the automatic loop detection can find annotation errors, which is a positive side-effect. Table VI shows the number of loops in the benchmarks and the number of loops where our DFA tool detected the bound. Four of the eight bounds could be detected for the kernel benchmarks; for the Mälardalen benchmarks, 32 out of 51 possible loop bounds were detected. For the application benchmarks, which are of course particularly interesting to judge the usefulness of the analysis, 19 out of 29 loops could be bounded. In total, almost two thirds of the loops could be successfully bounded.

For the benchmarks derived from industrial applications, the percentage of detected loop bounds is between 60% and 67%. It is notable that for the Kfl benchmark all loops that could not be bounded involve I/O or are actually infinite loops.

Although being a rather simple analysis, we found the loop bounds analysis described in Section 6 to be very useful in the context of automating the WCET analysis. With the DFA we found several

Table VI. Automatically detected loop bounds (detected/loops)

Program	Detected	Total
crc	1	1
LineFollower	0	0
SVM	3	7
BinarySearch	0	1
Bubble	3	3
Crc	3	3
ExpInt	3	3
FDCT	2	2
Fibonacci	1	1
InsertionSort	1	2
JanneComplex	0	2
MatrixCount	4	4
MatrixMult	5	5
NestedSearch	4	4
QuickSort	0	6
Select	0	4
SLE	6	11
Lift	4	6
Kfl	12	18
EjipCmp	3	5

annotation errors in our test programs. In the Kfl benchmark a few loops were annotated with a too high value. That error resulted in a slightly higher WCET estimate, but is otherwise harmless. More serious annotation bugs were found in the Lift and SVM benchmarks where the annotation was too low. These experiences are an argument to automate loop bound detection as much as possible.

7.2.3. Tools Runtime

The user acceptance of analysis tools depends also on the execution time. A tool that runs for days to provide results can be considered almost useless. Providing results within minutes is probably acceptable, giving feedback in the range of seconds allows integrating analysis tools in interactive development environments, like background compilation in Eclipse. Table VII gives the execution time of the WCET analysis and the DFA with the receiver and loop bound analysis. The execution time was measured on an Intel Core2 Duo notebook CPU running at 2.2 GHz.

The second and third columns in Table VII show the execution time. It is given in milliseconds for WCA and in seconds for DFA. The method local based WCET analysis with the static cache approximation results in very short execution time between 15 ms and 172 ms.

Table VII. Execution time of WCET analysis and DFA

Program	WCA (ms)	DFA (s)
crc	16	1.8
LineFollower	31	2.6
SVM	15	349
BinarySearch	15	2.0
Bubble	0	1.9
Crc	31	3.5
ExpInt	15	3.0
FDCT	31	3.0
Fibonacci	16	1.9
InsertionSort	15	2.0
JanneComplex	16	2.4
MatrixCount	15	3.1
MatrixMult	16	3.2
NestedSearch	16	87
QuickSort	32	3.6
Select	16	3.0
SLE	15	3.3
Lift	32	5.8
Kfl	172	20
EjipCmp	62	138

The receiver and loop bound analyses usually finish within a few seconds, which is very acceptable. However, SVM and NestedSearch require a disproportional amount of time. As mentioned before, SVM contains around 3000 lines of initialization code that are included in the DFA.

7.3. Model Checking Performance

We have evaluated the time needed to estimate the WCET using the UPPAAL model checker with a subset of the benchmarks. We measure the total time spent in the binary search, and the maximum time spent in one run of the UPPAAL verifier. All experiments were carried out using an Intel Core Duo 1.83 GHz with 2GB RAM on darwin-9.5, using UPPAAL 4.0.

Table VIII illustrates the connection between the number of inner loop iterations and the time needed for model checking. As UPPAAL generates states for each loop iteration, we have to expect that larger loop bounds lead to longer analysis times. The increase in the time needed for verification is roughly proportional to the WCET. The results also illustrate that eliminating the binary search would reduce the model checking time significantly.

In Table IX, the time needed for model checking embedded applications is listed. We measure both the time spent model checking the full problem and the locally simplified problem. The latter is

Table VIII. Investigating the connection between loop bounds and model checking time

Problem	N	WCET 10^6 cycles	Verification Time (s)	Binary Search Time (s)
Fibonacci ($O(N)$)	10000	0.3	0.4	6.4
Fibonacci	100000	3.9	3.3	66.0
Fibonacci	1000000	39.0	28.8	751.7
BubbleSort ($O(N^2)$)	100	1.8	0.5	9.7
BubbleSort	350	22.6	5.6	120.0
BubbleSort	1000	185.7	35.1	913.0
MatrixMult ($O(N^3)$)	25	2.1	0.4	7.6
MatrixMult	50	16.7	2.9	58.5
MatrixMult	100	134.0	22.3	511.3

Table IX. Analysis times for model checking based and IPET based WCET analysis of embedded applications

	UPPAAL		UPPAAL simplified		
Problem	Verifier (s)	Search (s)	Verifier (s)	Search (s)	IPET (s)
LineFollower	0.11	1.13	0.15	1.32	0.031
Lift	0.16	2.19	0.09	1.18	0.032
Kfl	0.63	9.44	0.41	6.57	0.172

obtained by first analyzing invocation-free inner loops and leaf methods separately, and then replacing them by pseudo locations annotated with the WCET of the program fragment. For comparison the last column shows the execution time of the IPET based WCET analysis (executed on a 2.2 GHz notebook).

In contrast to loops, complex control structures itself do not lead to long analysis times. But without additional state variables, there is no benefit of using model checking instead of the IPET approach. Global state is easy to model using UPPAAL, and leads to accurate hardware models, as shown in the next section. However, a large state space, for example caused by the method cache or global variables, may lead to a state space explosion, and may prohibit the verification.

7.4. Cache Analysis with Model Checking

To evaluate if the static cache approximation in the IPET based WCET analysis is tight enough, we compare it with the more expensive cache simulation in the model checking approach. In this experiment we assume a smaller method cache of 1 KB. Table X shows the WCET estimates for

Table X. Execution time in clock cycles for different approaches to cache analysis

Program	Single method IPET	FIFO cache IPET	FIFO cache UPPAAL
LineFollower	2610	2411	2368
Lift	8897	8595	8355
Kfl	49765	40473	37894

three settings: Assuming a cache where every access is a cache miss (Single method IPET),** using the static method cache approximation (FIFO cache IPET), and the UPPAAL cache simulation (FIFO cache UPPAAL).

When comparing the IPET based and model checking based cache analysis modeling, the method cache in UPPAAL results in a 2% to 7% lower WCET bound. However, the analysis runtime is one to two orders of magnitude higher with model checking. Therefore, we conclude that our static cache approximation for IPET is tight enough to be practical.

8. Discussion

We found that the approach of a time-predictable processor and static WCET analysis is a feasible solution for future safety critical systems. However, there are some remaining questions that are discussed in the following section. We explore the confidence in the correctness of the WCET analysis and how general the tools are.

8.1. On Correctness of WCET Analysis

Safe WCET approximation depends on the correct modeling of the target architecture and the correct implementation of the analysis tools. During the course of this paper we have found a bug in the WCA from 2006 and have also found bugs in the low-level timing description of JOP. However, the open-source approach for the processor and the tools increases the chance that such bugs are found. Furthermore, we have compared the results of the new implementation of WCA with two independent implementations of WCET analysis for JOP: The Volta project by Trevor Harmon [23] and the former implementation of WCA [68]. The results differ only slightly due to different versions of JOP that Volta used and a less accurate cache analysis implemented in the second tool. It should be noted that JOP and the WCA are still considered on-going research prototypes for real-time systems.

**Instruction access within a method are still guaranteed hits. The cache is just assumed to cache a single method.

8.2. Is JOP the Only Target Architecture?

We have primarily considered JOP as the low-level target for the WCET analysis as it is: (a) A simple processor, (b) open-source, and (c) the execution timing is well-documented. Furthermore, the JOP design is actually the root of a family of Java processors. Flavius Gruian has built a JOP compatible processor, with a different pipeline organization, with Bluespec Verilog [17]. The SHAP Java processor [79], although now with a different pipeline structure and hardware assisted garbage collection, also has its roots in the JOP design. We assume that these processors are also WCET analysis *friendly*. The question remains how easy the analysis can be adapted for other Java processors.

picoJava [72, 73], the Java processor from Sun, is probably too complex to model. The pipeline and the interaction between the different memories (prefetch buffer, stack buffer, and caches) will result in rather conservative WCET estimates.

The real-time Java processor jamuth [75] is a possible target for the WCA. There is on-going work with Sascha Uhrig to integrate the timing model of jamuth into the WCA. A preliminary version of that model has been tested. The analysis tool need to be changed to model a few pipeline dependencies.

Still, WCET analysis of Java programs on RISC processors with a compiling JVM is a challenge. With a compiling JVM, the WCET friendly bytecodes are further compiled into machine code. Information is lost in the second transformation, while usually performance is gained. This is not the case with a dedicated Java processor such as JOP.

8.3. Object-oriented Evaluation Examples

Most of the benchmarks used are written in a rather conservative programming style. The Mälardalen benchmarks are small kernels that do not stress the cache analysis. We would like to use more object-oriented real-time examples for the evaluation. Then, we can evaluate if object-oriented programming style with dynamic method dispatch and rather small methods is a feasible option for future real-time systems.

8.4. Programming Idioms for Loop Bound Analysis

While some of the undetected loop bounds for the benchmarks were simply algorithmically too complex for our analysis (e. g., a binary search on an array cannot be bounded), other loop bounds appeared to be undetectable due to certain programming idioms.

One operation that is difficult to handle for an interval based value analysis is equality/inequality: the operation has no useful definition on intervals. A condition $x \neq 0$ is true for $x \in ([-\infty, -1] \cup [1, \infty])$. Merging these two sub-intervals into a single interval again yields $[-\infty, \infty]$, i. e., no information has been gained from the operation. In contrast, less-than/greater-than operations in loop exit conditions help the analysis to find a bound, because they narrow down the possible values for the loop variable. A related issue is the fact that our loop bound analysis cannot handle boolean flags for loop exit conditions. When using a boolean flag, the analysis cannot detect a sensible increment, and therefore assumes that the loop is unbounded.

Some applications (especially the Kfl benchmark) use loops that depend on the environment, e. g., a timer reaching a certain value. Although it may be guaranteed that the program eventually exits the loop, the number of loop iterations until it does so is not specified inside the program logic. For

such cases, it is inevitable that there exists some mechanism for a developer to specify a loop bound manually.

8.5. WCET Analysis for Chip-multiprocessors

Chip-multiprocessors (CMP) are now considered as the future technology for high-performance embedded systems. Christof Pitter has built a CMP version of JOP that includes a time-predictable memory arbiter [48]. During the course of his thesis he has adapted the low-level timing calculation of the WCET tool to integrate the memory access times with the time-sliced arbiter. Therefore, it is possible to derive WCET values for tasks running on a CMP system with shared main memory. To the best of our knowledge, [48] presents the first time-predictable CMP system where WCET analysis is available.

The first estimation of bytecode timings for the CMP system considers only individual bytecodes. However, with variable memory access times due to memory arbitration this estimate is conservative. An extension to basic block analysis and using model checking to analyze possible interactions between the memory arbitration process and the program execution should result in tighter WCET estimates.

The JOP CMP system showed the importance of a tighter method cache analysis. Compared to the simple approach of the original WCA tool, the new static cache approximation yielded in 15% tighter WCET estimates for an 8 core CMP system.

8.6. Data Cache

In JOP only the stack content is stored in fast on-chip memory. Access to object and array fields goes directly to main memory. This setup is good enough for a uniprocessor solution with fast main memory (e.g., SRAM). However, for a CMP version of JOP the memory bandwidth becomes the bottleneck and we have added a data cache with high associativity and least-recently used (LRU) replacement policy to JOP. The high associativity is intended to allow WCET analysis of heap accesses where the addresses are not statically known.

As a first approach, we will include data cache analysis similar to the method cache analysis. Within loops the number of heap accesses to different addresses (different references plus field/array offsets) is determined. If that number is lower than or equal, the associativity we can qualify all heap accesses as a single miss for one loop iteration and hits for the other iterations [66]. With this type of analysis two replacement policies are feasible: LRU and FIFO. FIFO consumes a little fewer resources than LRU, but the main issue, with respect to resources and hit detection time, is the high associativity.

8.7. Co-Development of Processor Architecture and WCET Analysis

JOP is an example of a time-predictable computer architecture. An extension of this approach to RISC and very-long instruction word based processors is presented in [67]. We consider a co-development of time-predictable computer architecture and WCET analysis for the architectural features as an ideal approach for future real-time systems. A new time-predictable feature (e.g., a data cache organization) can only be evaluated when the analysis is available. The analysis, and what can be analyzed within

practical execution time and memory demands, shall guide the computer architect of future time-predictable processors.

8.8. Further Paths to Explore

We have not yet fully explored the benefits and limitations of data-flow analysis for Java to support the WCET analysis. The examples in the evaluation section are medium sized tasks, and the DFA sometimes needs considerable execution time. We have to explore the right tradeoff between analysis accuracy and execution time for larger sized applications. For example, a short experiment to exactly model the method cache with DFA leads to a state space explosion even for small example applications.

The combination of a bytecode based optimizer (a prototype is available as part of the JOP sources) with WCET analysis can be applied to minimize the WCET path. The optimizer can be guided by the results from the analysis. This iterative flow can be used to minimize the worst-case path instead of the average-case path in the application.

9. Conclusion

In this paper, we have presented the combination of a time predictable Java processor and a WCET analysis tool. The processor and the tool are open-source under the GNU GPL.[††] The architecture of the Java processor greatly simplifies the low-level part of the WCET analysis as it can be performed at the Java bytecode level. An instruction cache, named *method cache*, stores complete methods, and is integrated into the WCET analysis tool.

The WCET analysis tool, with the help of DFA detected loop bounds, provides WCET values for the schedulability analysis. Complex loops still need to be annotated in the source. A receiver type analysis helps to tighten the WCET bounds of object-oriented programs. We have also integrated the method cache analysis into the tool. The cache can be analyzed at the method level and does not need the full program CFG. Besides the calculation of the WCET bound, 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.

Furthermore, we have evaluated a model checking approach to WCET analysis. From the comparison of IPET with model checking based WCET analysis we see that IPET analysis outperforms model checking analysis with respect to the analysis time. However, model checking allows easy integration of complex processor models, such as the model of the method cache. As future work we will further explore the combination of IPET and model checking based analysis, e.g., to analyze local effects such as complex timing interaction of a chip-multiprocessor version of JOP and use IPET for the global analysis.

As future work we consider extensions on the data-flow analysis to detect loop bounds for triangular loops. Some infeasible paths are detected by the loop bound analysis, but not yet exposed to the tool

[††]The source can be downloaded via git: git clone git://www.soc.tuwien.ac.at/jop.git

chain. On the one hand, infeasible path information can be used to optimize a program; on the other hand, WCET analysis is more precise if certain paths can be excluded. A future revision of the analysis framework will therefore provide means to access this information.

ACKNOWLEDGEMENTS

The authors thank Peter Puschner for the discussion on ILP based WCET analysis and his constructive comments on earlier versions of the paper. We thank Anders P. Ravn for suggesting UPPAAL for the model checking based WCET analysis and the initial UPPAAL model of the method cache. Furthermore, we thank the anonymous reviewers for their detailed reviews of the initial paper which guided us to improve the paper.

REFERENCES

1. AbsInt Angewandte Informatik GmbH. *PAG The Program Analyzer Generator User's Manual*, 2008. Version 2.0.
2. R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, 1994.
3. 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.
4. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM.
5. 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.
6. I. Bate, G. Bernat, and P. Puschner. Java virtual machine support for portable worst-case execution time analysis. In *ISORC. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Washington, USA, Jan 2002.
7. G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
8. 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.
9. T. Bogholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen. Model-based schedulability analysis of safety critical hard real-time Java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 106–114, New York, NY, USA, 2008. ACM.
10. 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.
11. J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. J. Gil, and A. J. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 204–213, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.
12. C. Cullmann and F. Martin. Data-flow based detection of loop bounds. In C. Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
13. M. Dahm. Byte code engineering with the BCEL API. Technical report, Freie Universitat Berlin, April 2001.
14. B. Dobbing and A. Burns. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 annual ACM SIGAda international conference on Ada*, pages 1–6. ACM Press, 1998.
15. J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):437–455, Aug. 2003.
16. C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

17. F. Gruian and M. Westmijze. Bluejep: a flexible and high-performance java embedded processor. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 222–229, New York, NY, USA, 2007. ACM.
18. J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Uppsala University, 2000.
19. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 57–66, Washington, DC, USA, 2006. IEEE Computer Society.
20. T. R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, August 2000.
21. D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.
22. T. Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.
23. T. Harmon and R. Klefstad. Interactive back-annotation of worst-case execution time analysis for Java microprocessors. In *Proceedings of the Thirteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, August 2007.
24. T. Harmon and R. Klefstad. Toward a unified standard for worst-case execution time annotations in real-time java. In *Proceedings of the Fifteenth International Workshop on Parallel and Distributed Real-Time Systems*, Long Beach, California, March 2007. IEEE.
25. T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, St. Louis, MO, United States, April 2008.
26. C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. V. Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, 18:121–148, 2000.
27. C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, 1999.
28. C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, 1995.
29. R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.
30. T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. *Electronic Notes in Theoretical Computer Science*, 2009.
31. E. Y.-S. Hu, G. Bernat, and A. Wellings. Addressing dynamic dispatching issues in WCET analysis for object-oriented hard real-time systems. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2002*, pages 109–116, Apr 2002.
32. E. Y.-S. Hu, A. J. Wellings, and G. Bernat. Deriving java machine timing models for portable worst-case execution time analysis. In *On the Move to Meaningfull Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems*, volume LNCS 2889, pages 411–424. Springer, 2003.
33. J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime java programs. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, pages 162–169, New York, NY, USA, 2006. ACM Press.
34. Imsys. Im1101c (the Cjip) technical reference manual / v0.25, 2004.
35. R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel. WCET analysis: The annotation language challenge. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis*, Pisa, Italy, July 2007.
36. 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.
37. L. M. Kristensen and T. Mailund. A generalised sweep-line method for safety properties. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 549–567, London, UK, 2002. Springer-Verlag.
38. C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.
39. 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.
40. 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.

41. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
42. T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 12–21, Washington, DC, USA, 1999. IEEE Computer Society.
43. Mälardalen Real-Time Research Center. WCET benchmarks. Available at `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, accessed February 11, 2009.
44. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
45. J. M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.
46. M. Ouimet and K. Lundqvist. Verifying execution time using the TASM toolset and UPPAAL. Technical Report Embedded Systems Laboratory Technical Report ESL-TIK-00212, Embedded Systems Laboratory Massachusetts Institute of Technology.
47. R. U. Pedersen. Hard real-time analysis of two java-based kernels for stream mining. In *Proceedings of the 1st Workshop on Knowledge Discovery from Data Streams (IWKDDS, ECML PKDD 2006)*, Berlin, Germany, September 2006.
48. C. Pitter. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.
49. A. Prantl, J. Knoop, M. Schordan, and M. Triska. Constraint solving for high-level WCET analysis. In *The 18th Workshop on Logic-based methods in Programming Environments (WLPE 2008)*, pages 77–89, Udine, Italy, December 2008.
50. T. B. Preusser, M. Zabel, and R. G. Spallek. Bump-pointer method caching for embedded java processors. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, pages 206–210, New York, NY, USA, 2007. ACM.
51. W. Puffitsch. picoJava-II in an FPGA. Master's thesis, Vienna University of Technology, 2007.
52. 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.
53. P. Puschner and A. Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.
54. P. Puschner and A. Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society.
55. P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.
56. 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.
57. V. S. P. L. E. G. Raja Vallée-Rai, Laurie Hendren and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
58. J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007.
59. 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.
60. 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.
61. M. Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.
62. M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.
63. 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.
64. M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.
65. M. Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Number ISBN 978-1438239699. CreateSpace, August 2009. Available at `http://www.jopdesign.com/doc/handbook.pdf`.
66. M. Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.
67. M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.
68. M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM

Press.
69. A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng.*, 15(7):875–889, 1989.
70. O. Shivers. Control flow analysis in scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM.
71. O. Shivers. The semantics of scheme control-flow analysis. In *PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 190–198, New York, NY, USA, 1991. ACM.
72. Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.
73. Sun. *picoJava-II Programmer's Reference Manual*. Sun Microsystems, March 1999.
74. V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.
75. S. Uhrig and J. Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press.
76. A. Wellings. Is Java augmented with the RTSJ a better real-time systems implementation technology than Ada 95? *Ada Lett.*, XXIII(4):16–21, 2003.
77. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.
78. R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.
79. M. Zabel, T. B. Preusser, P. Reichel, and R. G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Lübeck, Germany, Aug. 2007.