COMPARISON OF IMPLICIT PATH ENUMERATION AND MODEL CHECKING BASED WCET ANALYSIS

Benedikt Huber and Martin Schoeberl[1]

Abstract

In this paper, we present our new worst-case execution time (WCET) analysis tool for Java processors, supporting both implicit path enumeration (IPET) and model checking based execution time estimation. Even though model checking is significantly more expensive than IPET, it simplifies accurate modeling of pipelines and 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 may lead to long analysis times. To obtain a tool which is able to cope with larger applications, we recommend to use model checking for more important code fragments, and combine it with the IPET approach.*

1. Introduction

Worst-case execution time (WCET) analysis is needed as input to schedulability analysis. As measuring the WCET is not a safe approach, real-time tasks have to be analyzed statically. We present a WCET analysis tool for Java. The first target processor that is supported by the tool is the Java processor JOP [24]. A future version will also support the multi-threaded Java processor jamuth [27].

The WCET analysis is performed on Java bytecodes, the intermediate representation of compiled Java programs. A Java virtual machine (JVM) executes bytecodes by either interpreting the bytecodes, compiling them to native code, or executing them in hardware. The latter case is the execution model of a Java processor. Using bytecode as the instruction set without further transformation simplifies WCET analysis, as the execution time can be modeled at the bytecode level.

The primary method for estimating the WCET is based on the standard IPET approach [21]. We have implemented a context sensitive, bottom-up analysis, and an interprocedural analysis to support a simple method cache approximation. The tool additionally targets the UPPAAL model checker [3]. The control flow graphs (CFG) are modeled as timed automata, and a binary search is used to obtain the WCET. An (exact) cache simulation was added to the UPPAAL model, which is used to evaluate the quality of static cache approximations. Having implemented those two methods in a single tool, we compare traditional IPET with model checking based techniques.

Although we analyze Java programs, neither the model checking based analysis nor the method cache are Java specific (e.g., the method cache has been implemented in the CarCore processor [15]).

[1]Institute of Computer Engineering, Vienna University of Technology, Austria;
email: `benedikt.huber@student.tuwien.ac.at`, `mschoebe@mail.tuwien.ac.at`

1.1. The Target Architecture JOP

JOP [24] is an implementation of the Java virtual machine (JVM) in hardware. The design is optimized for low and predictable WCETs. Advanced features, such as branch prediction or out-of-order execution, that are hard to model for the WCET analysis, are completely avoided. Instead, the pipeline and the caches are designed to deliver good real-time performance.

Caches are a mandatory feature for pipelined architectures to feed the pipeline with enough instructions and speedup load/store instructions. Although direct-mapped caches or set-associative caches with a least recently used (LRU) replacement policy are WCET friendly, the instruction cache in JOP has a different organization. The cache stores full methods and is therefore named method cache [23]. The major benefit of that cache is that cache misses can only occur at method invoke or on a return from a method. All other instructions are guaranteed hits and the cache can be ignored during WCET analysis.

JOP uses a *variable block method cache*, where methods are allowed to occupy more than one cache block. JOP currently implements the variable block cache using a so called next-block replacement, effectively a first in, first out (FIFO) strategy. It is known, that the FIFO replacement strategy is not ideal for WCET analysis [22]. A LRU replacement would be preferable, as it provides a 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 a LRU replacement strategy difficult.

1.2. Related Work

WCET related research started with the introduction of timing schemas by Shaw [26]. Shaw presents rules to collapse the CFG of a program until a final single value represents the WCET. An overview of actual WCET research can be found in [20, 28]. Computing the WCET with an integer linear programming solver is proposed in [21] and [13]. The approach is named graph-based and implicit path enumeration respectively. We base our WCET analyzer on the ideas from these two groups.

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 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 [1, 9] and cache analysis integration with the pipeline analysis [8]. Heckmann et. al described the influence of different cache architectures on WCET analysis [10]. Our approach to cache analysis is to simplify the cache with the method cache. That form of caching needs no integration with the pipeline analysis and the hit/miss categorization can be approximated at the call graph, which leads to shorter analysis times.

WCET analysis at the bytecode level was first considered in [4]. It is argued that the well formed intermediate representation of a program, the Java bytecode, 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 in [2] to address the machine-dependent low-level timing analysis. Worst-case execution frequencies of Java bytecodes are introduced for a machine independent timing information. Pipeline effects (on the target machine) across bytecode boundaries are modeled by a *gain time* for bytecode pairs. Due to our target architecture that executes Java bytecode natively we can extend on the work of WCET analysis at bytecode level.

In [25], an IPET based WCET analysis tool is presented that includes the timing model of JOP. 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 [6]. The tool is extended to integrate JOP's method cache [7] in a similar way as in [25]. Compared to those two tools, which also target JOP, our presented WCET tool is enhanced with: (a) analysis of bytecodes that are implemented in Java; (b) a tighter method cache analysis; and (c) an evaluation of model checking for WCET analysis and exact modeling of the method cache in the model checking approach.

Whether and to what extend model checking can deliver tighter WCET bounds than IPET is investigated in [16]. Though we use timed automata and a different cache model, the argument that state exploration is potentially more precise than IPET still applies. A project closely related to our model checking approach is presented in [5]. Model checking of timed automata is used to verify the schedulability of a complete application. However, even with a simple example, consisting of two periodic and two sporadic tasks, this approach leads to a very long analysis time.

2. The WCET Analysis Tool

The new WCET analysis tool deals with a less restricted subset of Java than [25], adding support for bytecodes implemented in Java and dynamic dispatch. The supported subset of the Java language is restricted to acyclic callgraphs. As any static WCET analysis, we require the set of classes to be known at compile time.

For determining loop bounds, we rely on both source code annotations (similar to the ones described in [11, 25]) and dataflow analysis [19]. The latter is also used for computing the set of methods possibly executed when a virtual method is invoked. By default, the WCET is calculated using IPET, taking the variable block method cache into account (see Section 2.1.). Additionally, a translation to timed automata has been implemented, including method cache simulations (see Section 2.2.).

2.1. IPET and Static Cache Approximation

The primary method for estimating the WCET is based on the standard IPET approach [21]. Hereby, the WCET is computed by solving a maximum cost circulation problem in a directed graph representing the program's control flow. Each edge is associated with a certain cost for executing it, and linear constraints are used to exclude infeasible paths.

In addition to the usual approach modeling the application as one big integer linear program (ILP), we also support a bottom-up analysis, which first analyses referenced methods separately and then solves the intraprocedural ILP model. This is a fast, simple alternative to the global model, and allows us to invoke the 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, we create detailed reports to provide feedback to the user and annotate the source code as far as possible. 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.

Computing the Cache Cost To compute the cost of cache misses for JOP, we observe that a cache miss either occurs when executing an invoke instruction or when control returns to the caller. Each method takes a fixed number of cycles to load (depending on its size), and depending on the instruction triggering the cache miss, a certain number of those load cycles are hidden.

FIFO caches show unbounded memory effects [14] and simulating the cache with an initially empty cache is not a safe approximation. This applies even when the cache sequence corresponds to a path in a non-recursive call tree, so it is necessary to *flush the cache* at the task's entry point to get a safe approximation using simulation. As furthermore a long access history is needed to classify a cache access as hit or as miss, standard dataflow analysis techniques perform poorly on FIFO caches [22].

Approximating the Method Cache using Static Analysis Our technique to approximate the cache miss cost for an N-block FIFO method cache is based on the following fact: If it is known that during the execution of some method m, at most N distinct cache blocks (including those of m) are accessed, each accessed block will be loaded *at most once* when executing m. Currently a simple heuristic is used to identify such *all-fit* methods, by checking whether all methods possibly invoked during the execution of m fit into the cache.

To include this approximation into the ILP model, we traverse the callgraph of the program starting from the root method, and check for each call site, whether the invoked method m is *all-fit*. If this is the case, the supergraph of m is duplicated, adding constraints that the code of each method possibly accessed during the execution is loaded at most once. Otherwise, the invoke and return cache access are considered to be a cache miss.

2.2. Calculating the WCET using Model Checking

UPPAAL is a model checker based on networks of timed automata, supporting *bounded integer variables* and synchronization using *channels* [3]. We have implemented a translation of (Java) programs to UPPAAL models, and use the model checker to determine a safe WCET bound. This allows us to potentially deal with complex timing dependencies, and gives us the opportunity to verify whether and to what extent model checking works for practical WCET estimation. Our initial attempt was based on ideas from [17, 5] and has been subsequently refined using progress measures and adding cache simulations.

General Strategy An UPPAAL model comprises global declarations of clocks, synchronization channels and 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 timed automata 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 I, which is the entry point of the program, and one location E, which corresponds to the program's exit. When execution has finished, the system takes the transition from E to the final state EE (Figure 1). 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 E, $t \leq t_{guess}$ holds. If this property is satisfied, t_{guess} is a safe WCET bound, otherwise we have to assume it is not. Starting with a known

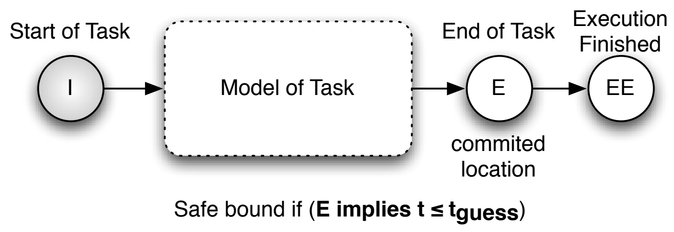
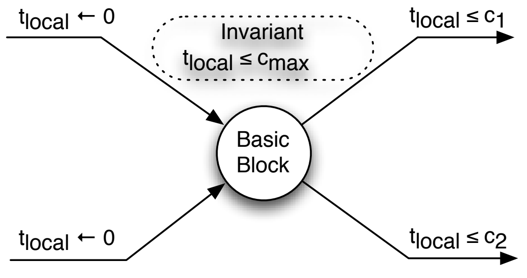
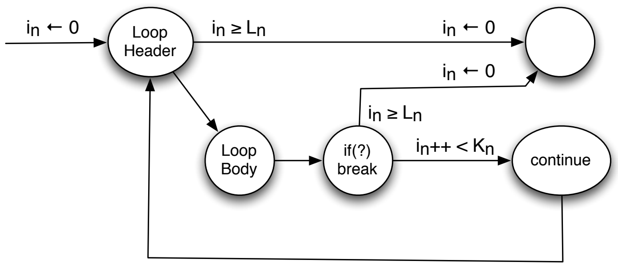


Figure 1. Calculating the WCET bound using UPPAAL



(a) Modeling basic blocks



(b) Modeling loops

Figure 2. Modeling CFGs as Timed Automata

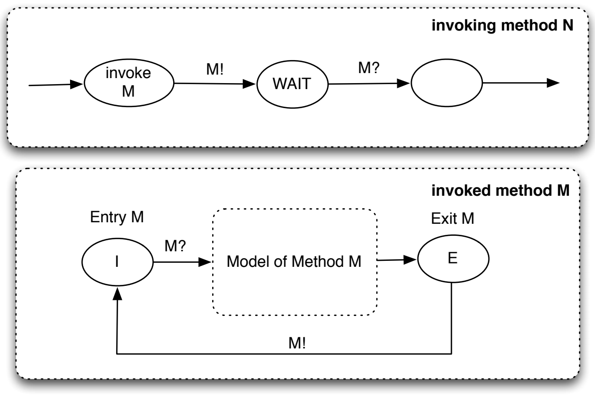
upper bound, provided by the IPET analysis, we perform a binary search to find a tighter WCET bound. Note that if the model checker kept track of the maximum value of t encountered during state exploration, it would not be necessary to perform a binary search.

Translation of CFGs Given the CFG of a Java method m_i, we build an automaton 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 $M_i.I$ corresponds to the entry node of the CFG, and the location $M_i.E$ to its exit node.

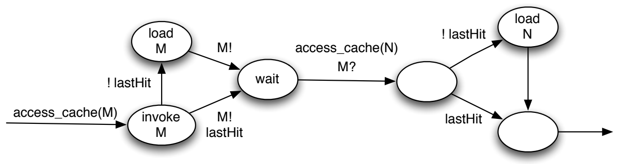
To model the timing of basic blocks, we reset t_{local} at the incoming edges of a basic block. If the execution of the basic block takes at most c_{max} cycles, we add the invariant $t_{local} \leq c_{max}$ to the corresponding location. On architectures where the maximum number of cycles depends on the edge e taken, additional guards of the form $t_{local} \leq c_e$ are added to the outgoing edges of the basic block (Figure 2(a)).

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 K_n times. We declare a local bounded integer variable i_n representing the loop counter, ranging from 0 to K_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 $i_n < K_n$ 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 2(b)).

It might be beneficial to set $L_n = K_n$, but this is only correct in the absence of timing anomalies, and therefore in general unsound in the presence of FIFO caches. In principle, every control flow repre-



(a) Method Invocations



(b) Cache Simulation

Figure 3. Translating Method Invocations and Cache Accesses

sentable using bounded integer variables can be modeled using UPPAAL, though we only implemented simple loop bounds in the current version of our tool.

Method Invocations We build one automaton M_i for each reachable method m_i. To model method invocations, we synchronize the method's automata using channels. When a method m_i is invoked, the invoke transition synchronizes with the outgoing transition of $M_i.I$ on the invoked method's channel. When returning from method m_i, the transition from $M_i.E$ to $M_i.I$ synchronizes with the corresponding return transition in the calling method. This translation assumes that there are no recursive calls. To allow the method to be invoked several times, a transition from $M_i.E$ to $M_i.I$ is added to all methods (see Figure 3(a)).

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 3(b) and Listing 1).

```
const int NBLOCKS[NMETHODS] = { /* number of blocks per method */ };
const int UNDEF = NMETHODS;
/* In the initial state, the main method occupies the first, say, K blocks of
   the cache. Therefore, all fields of the array are undefined, except the one
   at position K−1, which is set to the id of the main method. */
int [0,NMETHODS] cache[NBLOCKS] = { UNDEF, ..., MAIN_ID, UNDEF, ... };
bool lastHit;                       /* whether last access was cache hit */
void access_cache(int mid) {
    int p = 0;                      /* pointer into the cache */
    int mblocks = NBLOCKS[mid];     /* blocks of accessed method */
    lastHit = false;                /* no cache hit so far */

    /* Check if mid is in the cache */
    for(p = 0; p < NBLOCKS; p++)
        if(cache[p] == mid) { lastHit = true; return; }

    /* Move cache blocks and insert new tag */
    for(p = NBLOCKS − 1; p >= mblocks; p−−)
        cache[p] = cache[p − mblocks];
    for(p = 0; p < sz−1; p++)
        cache[p] = UNDEF;
    cache[p] = mid;                 /* tag is written at position mblocks − 1 */
}
```

Listing 1. FIFO variable block cache simulation

Progress Measures and Optimizations The performance of the model checker crucially depends on the search order used for state exploration. Instead of using the default breadth-first search, performance is greatly improved by using an *optimal search order*. For a loop-free CFG, it is obviously beneficial to deal with node *a* before node *b*, if *a* is executed before *b* on each possible execution path. This topological order is generalized to CFGs with loops by taking the values and bounds of loop counters into account. Finally, we extend this idea to applications with more than one method by defining a relative *progress measure* [12], which is incremented on transitions and *monotonically increases* on each possible execution path. The progress measure both guides UPPAAL's state space exploration and reduces memory usage.

Additionally, for cache simulation purposes we pre-calculate the WCET of inner loops and leaf methods, using either model checking or IPET. The corresponding subgraphs of the CFG are replaced by summary nodes, resulting in a locally simplified model. Currently, we are developing a model checker prototype to explore further optimizations using state abstractions, optimized data structures, and exploiting sharing, caching, and locality.

3. Evaluation

In this section, we will present some experimental results obtained with our tool. The first problem set[2] consists of small benchmarks to evaluate the model checker's performance for intraprocedural analysis. The second set comprises benchmarks extracted from real world, embedded applications. Those examples are reactive systems, with many control flow decisions but few loops, making it difficult to obtain a good execution time estimate using measurements.

Table 1 lists the number of methods, the bytecode size of the tasks under consideration, and the number of control flow nodes in the (conceptually) unrolled supergraph. Additionally, we list the size of the largest method in bytes, which determines the minimal cache size. As WCET analysis targets

[2]Provided by the Mälardalen Real-Time Research Center, except GCD.

Problem	Description	Methods	Size (Total / Max)	Nodes
DCT	Discrete Cosine Transform (8×8)	2	968 / 956	45
GCD	Greatest Common Divisor (32 bit)	3	138 / 100	9599
MatrixMult	Matrix Multiplication (50×50)	3	106 / 84	19460
CRC	Cyclic Redundancy Check (40 bytes)	6	404 / 252	26861
BubbleSort	Bubble Sort (500×500)	2	87 / 80	1000009
LineFollower	A simple line-following robot	9	226 / 76	96
Lift	Lift controller	13	1206 / 216	438
UdpIp	Network benchmark	28	1703 / 304	9600
Kfl	*Kippfahrleitung* application	46	2539 / 1052	11348

Table 1. Problem sets for evaluation

Problem	Measured ET	Single Block	FIFO Cache	FIFO Cache	Pessimism
	JOP	IPET	IPET	UPPAAL	Ratio
DCT	19124	19131	19131	19124	1.00
GCD	62963	75258	73674	73656	1.17
MatrixMult	1.09M	1.09M	1.09M	1.09M	1.00
CRC	0.19M	0.47M	0.38M	0.38M	2.00
BubbleSort	32.16M	46.33M	46.33M	46.33M	1.44
LineFollower	2348	2610	2411	2368	1.03
Lift	5484	8897	8595	8355	1.57
UdpIp	8375	131341	130518	129638	15.58
Kfl (8 Block)	10616	49744	40452	37963	3.81

Table 2. Measured and computed WCET

single tasks, the size of the considered applications seems to be realistic for embedded systems. On the other hand, we would definitely benefit from a larger and varying set of benchmarks.

3.1. Comparison of Measured Execution Time and Calculated WCET

Table 2 compares the measured execution times and the computed WCET estimates. For all experiments, we use a memory access timing of 2 cycles for a read and 3 cycles for a write. We use a 16-block variable-block FIFO method cache, with 1 KB instruction cache memory in total.

The WCET was computed assuming a cache in which every access is a cache miss (Single Block IPET), using the static method cache approximation described in Section 2.1. (FIFO Cache IPET), and the UPPAAL cache simulation presented in the last section (FIFO Cache UPPAAL). [3]

Pessimistic estimations in the first problem set are mainly due to missing flow facts and data dependent flow (`BubbleSort`, `CyclicRedundancyCheck`), while in the second problem set the measurements do not cover all execution paths (`Lift`, `Kfl`) or use a small payload (`UdpIp`). The estimates using the static cache approximation are quite close to the exact cache simulation using UPPAAL, so here, the approximation worked well. One should note, however, that the cache costs are in general small, and a bigger difference could occur on other platforms.

[3]For the `Kfl` benchmark, UPPAAL ran out of memory, so we had to reduce the number of cache blocks to 8.

Problem	IPET	UPPAAL Breadth First		UPPAAL Progress	
		Verify	Search	Verify	Search
DCT	0.00	0.09	1.21	0.07	0.84
GCD	0.00	3.10	47.65	0.20	2.75
MatrixMult	0.01	0.21	3.52	0.23	4.58
CRC	0.01	1.21	16.32	0.52	9.46
BubbleSort	0.00	7.63	197.91	10.33	268.43
LineFollower	0.00	0.11	1.07	0.15	1.10
Lift	0.01	0.17	2.07	0.18	1.38
UdpIp	0.03	8.98	84.69	1.78	30.28
UdpIp simplified		0.64	7.28	0.44	7.07
Kfl (no cache)	0.04	92.19	1229.42	0.57	9.23
Kfl simplified (no cache)		33.81	444.73	0.45	7.23
Kfl (8 blocks)	0.13	—	—	31.77	428.24
Kfl simplified (8 blocks)		—	—	17.72	263.18

Table 3. Analysis execution time in seconds

3.2. Performance Analysis

We have evaluated the time needed to estimate the WCET using the techniques presented in this paper, summarized in Table 3. For the approach using the UPPAAL model checker, we consider the translation without (UPPAAL Breadth First) and with progress measures (UPPAAL Progress Measure). For the two largest problems, we additionally consider locally simplified models (Section 2.2.). The time spent in the binary search and the maximum time needed for one invocation of the UPPAAL verifier are listed in the table. For comparison, we measure the time spent in the ILP solver, when using the global IPET approach.

All experiments were carried out using an Intel Core Duo 1.83 Ghz with 2 GB RAM on `darwin-9.5`. For IPET, the linear programs were solved using `lp_solve` 5.5. We use UPPAAL 4.0.7 with aggressive space compression (`-S 2`), but without convex hull approximation, as for our translation, the number of generated states is not affected by this choice.

For the first problem set, the analysis times for the IPET approach are below 10 milliseconds, as loops can be encoded efficiently using linear constraints. We observe that UPPAAL handles the analysis of those small benchmarks well too, as long as the number of executions of the innermost loop's body do not get too large (as in `BubbleSort`).

In the second problem set, the solver times for the IPET approach are still below 0.2 seconds, so the additional equations for the cache approximation did not result in a significant increase of the time needed for the ILP solver. Although model simplifications reduced the analysis times for the larger two problems, the most important optimization for UPPAAL is the use of progress measures. Progress measures can greatly improve performance (up to -99%), sometimes leading to slightly longer analysis times ($+35\%$). Furthermore, the cache simulation of the `Kfl` benchmark was only possible using progress measures, although we had to reduce the number of cache blocks to avoid a state explosion.

4. Conclusion

In this paper, we presented a WCET analysis tool for Java processors. The tool is open-source under the GNU GPL license.[4] The tool includes a static analysis of the effects of the method cache. The first target is the Java processor JOP. In a future version we will include jamuth as a second target.

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 hardware models, such as that of the method cache. We conclude that a combined approach where model checking is used on simplified problems delivers tight WCET bounds at a reasonable analysis time. In the current version of the tool we simplified CFGs and used model checking for the global analysis.

As future work we consider using model checking to analyze local effects and solve the global analysis with IPET. As an application of this approach we consider model checking for the complex timing interaction of a chip-multiprocessor version of JOP [18].

Acknowledgement

One author thanks Anders P. Ravn for suggesting UPPAAL for the model checking based WCET analysis during an enjoyable research visit at the University of Aalborg. We thank Anders for the initial UPPAAL model of the method cache and constructive comments on preliminary versions of the paper. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 214373 (Artist Design).

References

[1] ARNOLD, R., MUELLER, F., WHALLEY, D., AND HARMON, M. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium* (1994), pp. 172–181.

[2] BATE, I., BERNAT, G., MURPHY, G., AND PUSCHNER, P. Low-level analysis of a portable Java byte code WCET analysis framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications* (Dec. 2000), pp. 39–48.

[3] BEHRMANN, G., DAVID, A., AND LARSEN, K. G. A tutorial on Uppaal. In *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* (September 2004), M. Bernardo and F. Corradini, Eds., no. 3185 in LNCS, Springer–Verlag, pp. 200–236.

[4] BERNAT, G., BURNS, A., AND WELLINGS, A. Portable worst-case execution time analysis using Java byte code. In *Proc. 12th EUROMICRO Conference on Real-time Systems* (Jun 2000).

[5] BOGHOLM, T., KRAGH-HANSEN, H., OLSEN, P., THOMSEN, B., AND LARSEN, K. G. 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)* (New York, NY, USA, 2008), ACM, pp. 106–114.

[4] The source is available on the project's website `http://www.jopdesign.com`.

[6] HARMON, T., AND KLEFSTAD, R. 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).

[7] HARMON, T., SCHOEBERL, M., KIRNER, R., AND KLEFSTAD, R. 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).

[8] HEALY, C. A., ARNOLD, R. D., MUELLER, F., WHALLEY, D. B., AND HARMON, M. G. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers 48*, 1 (1999), 53–70.

[9] HEALY, C. A., WHALLEY, D. B., AND HARMON, M. G. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium* (1995), pp. 288–297.

[10] HECKMANN, R., LANGENBACH, M., THESING, S., AND WILHELM, R. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE 91*, 7 (Jul. 2003), 1038–1054.

[11] HU, E. Y.-S., KWON, J., AND WELLINGS, A. J. XRTJ: An Extensible Distributed High-Integrity Real-Time Java Environment. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications RTCSA-2003* (February 2003), vol. LNCS 2968, pp. 208–228.

[12] KRISTENSEN, L. M., AND MAILUND, T. 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* (London, UK, 2002), Springer-Verlag, pp. 549–567.

[13] LI, Y.-T. S., AND MALIK, S. 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* (New York, NY, USA, 1995), ACM Press, pp. 88–98.

[14] LUNDQVIST, T. *A WCET analysis method for pipelined microprocessors with cache memories*. PhD thesis, Chalmers University of Technology, Sweden, 2002.

[15] METZLAFF, S., UHRIG, S., MISCHE, J., AND UNGERER, T. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proceedings of the 9th workshop on Memory performance (MEDEA 2008)* (New York, NY, USA, 2008), ACM, pp. 38–45.

[16] METZNER, A. Why model checking can improve WCET analysis. In *Computer Aided Verification (CAV)* (Berlin/Heidelberg, 2004), vol. 3114 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 334–347.

[17] OUIMET, M., AND LUNDQVIST, K. Verifying execution time using the TASM toolset and UPPAAL. Tech. Rep. Embedded Systems Laboratory Technical Report ESL-TIK-00212, Embedded Systems Laboratory Massachusetts Institute of Technology.

[18] PITTER, C. *Time-Predictable Java Chip-Multiprocessor*. PhD thesis, Vienna University of Technology, Austria, 2009.

[19] PUFFITSCH, W. Supporting WCET analysis with data-flow analysis of Java bytecode. Research Report 16/2009, Institute of Computer Engineering, Vienna University of Technology, Austria, February 2009.

[20] PUSCHNER, P., AND BURNS, A. A review of worst-case execution-time analysis (editorial). *Real-Time Systems 18*, 2/3 (2000), 115–128.

[21] PUSCHNER, P., AND SCHEDL, A. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems 13*, 1 (Jul. 1997), 67–91.

[22] REINEKE, J., GRUND, D., BERG, C., AND WILHELM, R. Timing predictability of cache replacement policies. *Journal of Real-Time Systems 37*, 2 (Nov. 2007), 99–122.

[23] SCHOEBERL, M. 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)* (Agia Napa, Cyprus, October 2004), vol. 3292 of *LNCS*, Springer, pp. 371–382.

[24] SCHOEBERL, M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture 54/1–2* (2008), 265–286.

[25] SCHOEBERL, M., AND PEDERSEN, R. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)* (New York, NY, USA, 2006), ACM Press, pp. 202–211.

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

[27] UHRIG, S., AND WIESE, J. 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)* (New York, NY, USA, 2007), ACM Press, pp. 230–237.

[28] WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys. 7*, 3 (2008), 1–53.