A Framework to Quantify the Overestimations of Static WCET Analysis*

Hugues Cassé, Haluk Ozaktas, and Christine Rochange

University of Toulouse, France
`{casse,ozaktas,rochange}@irit.fr`

Abstract

To reduce complexity while computing an upper bound on the worst-case execution time, static WCET analysis performs over-approximations. This feeds the general feeling that static WCET estimations can be far above the real WCET. This feeling is strengthened when these estimations are compared to measured execution times: generally, it is very unlikely to capture the worst-case from observations, then the difference between the highest watermark and the proven WCET upper bound might be considerable. In this paper, we introduce a framework to quantify the possible overestimation on WCET upper bounds obtained by static analysis. The objective is to derive a lower bound on the WCET to complement the upper bound.

1998 ACM Subject Classification C.3 Computer Systems Organization – Special-Purpose and Application-Based Systems – Real-Time and Embedded Systems

Keywords and phrases static WCET analysis, uncertainty, overestimation, cache analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2015.1

1 Introduction

Tasks in hard real-time systems are subject to strict deadlines and any failure to meet a deadline might have severe consequences. Many approaches to hard real-time task scheduling have been proposed in the literature [4]. Based on the knowledge of the worst-case execution time (WCET) of individual tasks, they compute a task schedule that is valid in the sense that all deadlines are met.

The correctness of a task schedule relies on the confidence in the estimated WCETs of tasks. An under-estimated WCET might lead to an unsafe schedule and possibly to a violation of deadlines at runtime. Therefore, considering reliable techniques to derive those WCET estimates is crucial. The flexibility of measurement-based techniques is appealing but they cannot bring strict guarantees that the worst case has been observed or can be extrapolated. Static analysis techniques instead provide reliable estimates as soon as any detail on the hardware architecture of the target processor is known and correctly modelled[1].

Static analysis approaches determine an upper bound on the WCET based on an over-approximation of the processor state at any point in the program. For example, usual strategies to analyse the dynamic behaviour of a cache memory are based on abstract interpretation techniques that build abstract states of the cache at any program point.

By nature, WCET upper bounds are pessimistic and static WCET analysis techniques are often blamed for generating excessive overestimation. This feeling is exacerbated when

* This work was partially supported by the French research foundation (ANR) through the W-SEPT project (ANR-12-INSE-0001). See `http://wsept.inria.fr`.

[1] We are aware that this is a strong assumption that is difficult to achieve but this issue is beyond the scope of this paper.

the largest observed (measured) execution time is compared to the statically-determined WCET. The difference between the two is often exhibited as an indicator of overestimation although without any evidence (even any conviction) that the largest observed execution time was measured on the longest possible path. In the same time, it can generally not be shown that the estimated WCET value could be really reached. The goal of this paper is to show how additional numbers could be delivered together with the trustworthy WCET estimation in order to give better insight into how imprecise it might be.

Three factors can contribute to overestimating the WCET: (a) the analysis identifies a longest path that is not feasible in practice due to restrictions on input data values which are unknown (unspecified) to the analysis and/or to the incapacity of the analysis to take such restrictions correctly into account; (b) the analysis assumes a worst-case initial hardware state which cannot be observed in practice; (c) the way the analysis is performed generates overestimation (generally to reduce complexity). In this paper, we focus on factor (c) and leave the study of the two others for future work.

Our objective is to isolate the part of the estimated WCET that does not result of any over-approximation. We refer to this value as *a lower bound on the real WCET*: it represents an execution time that *is feasible* (assuming that both the sets of possible paths and initial hardware states are not over-approximated). The difference between the lower and upper bounds on the WCET gives an insight into the accuracy of the estimated WCET.

The paper is organised as follows: Section 2 introduces a framework to determine lower bounds along with upper bounds on the WCET of a task. A possible use of this framework for the cache analysis is presented in Section 3. Experimental results are reported in Section 4 and Section 5 concludes the paper.

2 A Framework to Quantify Pessimism of Static WCET Estimations

2.1 Static WCET Analysis Techniques

The usual flow for static WCET analysis is a sequence of several steps. The input is the program to analyse, more precisely its binary code required for low-level analyses. Its source code can be useful for path analyses.

In the first step (*Flow/Path Analysis*), possible execution paths are identified and encoded as a CFG (Control Flow Graph) and a set of flow facts (loop bounds, infeasible paths, etc.).

The next step (*Global Low-level Analysis*) determines the behaviour of history-based hardware schemes, such as (instruction and data) caches or branch predictors. These schemes make use of limited-capacity storage which may engender conflicts between distant parts of the program. In addition, they have been designed to improve the average execution time of the program by exploiting the execution history and hence may exhibit highly dynamic behaviours. These effects are particularly interesting to handle with static analysis approaches because time-expensive behaviours are relatively rare (but must be considered for safety reasons) and are particularly difficult to capture through measurements.

The third step (*Local Low-level Analysis*) takes the global and local behaviours of the hardware components into account to compute the possible execution times of each basic block in the CFG.

Finally, an ILP (Integer Linear Programming) system is built [10] for *WCET Computation* (IPET method). Its objective is to maximize the task execution time expressed as the sum of the basic blocks execution times weighted by their respective execution counts. These execution counts are bounded by a collection of linear constraints that express: (a) restrictions on the possible execution paths in the code, and (b) restrictions on the possible occurrences of some hardware-level behaviours.

2.2 Sources of Overestimation

The overestimation of static WCET analyses is due to three kinds of reasons:

- imprecise flow facts: in the absence of full information on possible execution paths, the WCET analysis might consider infeasible paths and those might happen to exhibit longer execution times than valid paths. The goal of flow analysis [14] is to eliminate such infeasible paths but it might fail to identify all of them for several reasons: (a) restrictions on input data values are under-specified by the user; (b) some information given by the user cannot be translated into flow facts (this depends on the annotation format considered by the WCET analysis tool, complex scenarii might be difficult to describe); (c) some flow facts derived from user annotations or automatically extracted from the source or binary code cannot be exploited during the analysis (e.g. they cannot be turned into linear constraints when the WCET is computed using the IPET method). In this paper, we assume that the applications under analysis do not suffer such issues and we consider that every relevant information of possible execution flows is known and taken into account when computing WCETs. We leave the analysis of imprecision due to incomplete information on input data for future work.
- imprecise information on the initial hardware state: the state of the processor (pipeline) and of the memory hierarchy (cache memories) has an impact on the (worst-case) execution time of a task. For example, if part of the task code already resides in the instruction cache, the overall latency of instructions fetches is shorter than if the cache is empty. In the absence of such information, the WCET analysis systematically assumes the worst-case situation and this might lead to overestimating execution times [12]. We ignore this issue in this paper and we assume that the tasks under analysis can effectively start with the worst-case hardware state.
- static analysis techniques do not unroll any possible execution path. Instead, they derive at each point in the code some properties that hold for any possible execution. This keeps the complexity of determining the longest path tractable. A consequence is that these properties might be pessimistic for a particular execution. Here, the pessimism is generated by the analysis technique and not by incomplete information on possible execution scenarii. The contribution of this paper is to quantify this overestimation which is inherent to static WCET analysis.

2.3 Proposed Framework

According to the discussion above, the execution time of an application code running on a given platform that ensures full isolation (i.e. the execution time of a task cannot be impacted by any other task or system operation) is a function of the initial hardware state and of the followed execution path:

$$\mathsf{ET} : \mathcal{H} \times \mathcal{P} \to \mathbb{N}$$

where $\mathcal{H}$ is the set of possible initial platforms states and $\mathcal{P}$ is the set of possible execution paths in the code.

The WCET of the program is defined as:

$$\mathsf{WCET} = \max_{(h,p)\in\mathcal{H}\times\mathcal{P}} ET(h,p)\,.$$

Static WCET analysis aims at determining an upper bound on the execution time. Instead of running the code with different $(h, p) \in \{\mathcal{H} \times \mathcal{P}\}$ pairs, as measurement-based approaches would do, it performs abstractions to derive, at each program point, some properties that hold for any execution path. These abstractions may lead to some undecided local behaviours which are then over-approximated to compute the final WCET estimation. For example, it can happen that the latency of an access to a cache is unknown because the cache analysis can not determine whether it will be a hit or a miss (because this depends on the execution history). One of these options must be chosen to compute the WCET. We use the term *scenario* to refer to a combination of selected options for all the unpredictable behaviors in the program and $\mathcal{C}$ denotes the set of all the possible scenarii.

The estimated (abstract) WCET can then be defined as follows:

$$\mathsf{WCET}^{\#} : \mathcal{H} \times \mathcal{P} \times \mathcal{C} \to \mathbb{N} .$$

Each set $\mathcal{D}$, where $\mathcal{D}$ stands for $\mathcal{H}$, $\mathcal{P}$ or $\mathcal{C}$, can be considered in several flavours:

- $\mathcal{D}^\top$ is the set of theoretically possible values.
  For example, $\mathcal{P}^\top$ is the set of possible paths expressed by the program control flow graph.
- $\mathcal{D}^+$ contains all the values that *may* (but are not guaranteed to) be feasible.
  In other words, $\mathcal{D}^+$ is a subset of $\mathcal{D}^\top$ that excludes the values that can be proven infeasible. For example, $\mathcal{P}^+$ includes all the execution paths that fulfil the flow constraints (both those computed by the flow analysis step and those specified by the user). Similarly, $\mathcal{C}^+$ is the set of all the scenarii that may occur and is considered by static WCET analysis to compute a trustworthy upper bound on the WCET.
- $\mathcal{D}^r$ is the set of all the values that *are really feasible.*
  Usually, this set is unknown or too large to be exhaustively explored.
- $\mathcal{D}^-$ is a set of values that *can be guaranteed to be feasible.*
  To illustrate this, let us consider an access to the cache that cannot be classified as a hit or a miss by the static cache analysis. If the processor is free of timing anomalies, the optimistic option is to consider the access as a hit, while the pessimistic assumption is a miss. Any scenario in $\mathcal{C}^-$ specifies a hit, and any scenario in $\mathcal{C}^+$ specifies a miss.

We have: $\mathcal{D}^- \subseteq \mathcal{D}^r \subseteq \mathcal{D}^+ \subseteq \mathcal{D}^\top$.

Usual static WCET analysis techniques produce an upper bound on the real WCET: $\mathsf{WCET}^+ : \mathcal{H}^+ \times \mathcal{P}^+ \times \mathcal{C}^+ \to \mathbb{N}$. Provided $\mathcal{H}^+$ and $\mathcal{P}^+$ are trustworthy, and assuming that the low-level analyses are correct (then $\mathcal{C}^+$ is trustworthy too), we must have $\mathsf{WCET}^r \leq \mathsf{WCET}^+$ where WCET^r is the real (but unknown) WCET of the application.

On the other hand, given that all the values in $\mathcal{H}^-$, $\mathcal{P}^-$ and $\mathcal{C}^-$ are feasible, it must be that $\mathsf{WCET}^- \leq \mathsf{WCET}^r$ with: $\mathsf{WCET}^- : \mathcal{H}^- \times \mathcal{P}^- \times \mathcal{C}^- \to \mathbb{N}$.

To summarize, $\mathsf{WCET}^- \leq \mathsf{WCET}^r \leq \mathsf{WCET}^+$. The possible overestimation on the real WCET is then quantified by $U = (\mathsf{WCET}^+ - \mathsf{WCET}^-)/\mathsf{WCET}^-$. We refer to WCET^- as the *lower bound on the WCET*[2] in contrast to WCET^+, the *upper bound on the WCET*. Only WCET^+ is a reliable estimation of the program's WCET.

Note that computing the WCET from sets of different flavours, e.g. $\mathcal{H}^- \times \mathcal{P}^- \times \mathcal{C}^+$, does not provide any useful estimate: it cannot be ordered with respect to the real WCET.

In this paper, we focus on the computation of $\mathcal{C}^-$ and $\mathcal{C}^+$ for various low-level analyses.

[2] Note that this lower bound on the WCET (WCET^-) is something different from the best-case execution time (BCET).

3 Upper-Bounding the Possible Overestimation of Low-Level Analyses

3.1 Background on Cache Analysis

Category-Based Approaches

Global low-level analyses account for the impact of history-based hardware schemes. Such schemes include instruction or data caches, or dynamic branch predictors. They offer either a fast access (a *hit* in the usual cache terminology), or a slow access (a *miss*), and are designed to maximize the number of *hits* in the average case.

A common approach to support such schemes is to assign a category to each access to the device. It has been successfully applied to LRU[3] instruction [7] and data [8] caches but also to branch prediction history tables [3] and flash memory prefetchers [5].

Instruction Cache Analysis

In [7] an analysis for LRU instruction caches introduces three categories: *Always-Hit* (AH) – on each access, the instruction block is present in the cache, resulting in a fast access; *Always-Miss* (AM) – the instruction block is never in the cache, always causing a slow access; and *Not Classified* (NC) – the behaviour cannot be predicted at analysis time. These categories have later been extended in [6, 1] with the *Persistence* category ($PERS$) that is assigned when the first occurrence of an access belonging to a loop body cannot be classified but the following accesses (in the next iterations) are *hits*.

The computation of categories is based on Abstract Interpretation (AI) techniques. A cache state is derived at each point of the program, as a function that assigns an age to each cache block: $\mathcal{S} : \mathcal{B} \to \mathcal{A}$. The age represents the position of the block in a cache set according to the LRU replacement policy: it is an integer value between 0 and A, where A is the associativity of the cache – an age equal to A means that the block is not in the cache. The *Update* function expresses the behaviour of the cache: $\mathcal{U} : \mathcal{S} \times \mathcal{B} \to \mathcal{S}$ (the accessed block gets age 0, the age of blocks younger than it is incremented by 1, the age of other blocks remains unchanged). To avoid an explosion of the number of possible cache states at a program point, abstract interpretation merges states using a *Join* function $\mathcal{J}^{\#} : \mathcal{S}^{\#} \times \mathcal{S}^{\#} \to \mathcal{S}^{\#}$ that operates on an abstract representation $\mathcal{S}^{\#}$ of the concrete cache states.

A category is assigned to an access to cache block $b \in \mathcal{B}$ at any program point p as follows: (i) if any state $s \in \mathcal{S}$ contains b whatever the path leading to p, the access always hits (AH); (ii) if no path leading to p produces an $s \in \mathcal{S}$ that contains b, then the access always misses (AM). Two abstract interpretation analyses are needed to draw such conclusions: (i) the MUST analysis computes the worst-case age of block b (if the worst-case age of b is less than A at point p, this means that b is always in the cache at point p) with $\mathcal{J}^{\#}(s_1, s_2) = s$ such that $\forall b \in \mathcal{B}, s[b] = max(s_1[b], s_2[b])$; and (ii) the MAY analysis computes the best-case age of b (if the best-case age of b is A at point p, this means that b is never in the cache at point p) with $\mathcal{J}^{\#}(s_1, s_2) = s$ such that $\forall b \in \mathcal{B}, s[b] = min(s_1[b], s_2[b])$.

If none of the two previous predicate holds, the access is considered as NC[4]. This category is the main source of overestimation in the WCET estimation.

[3] LRU stands for Least Recently Used and refers to the cache block replacement policy. We consider this policy in the remainder of the paper.

[4] As mentioned above, another category for blocks that remains in the cache across the iterations of a loop but cannot be classified for their first access can be assigned. This analysis is not described here for the sake of simplicity. The reader may refer to [6, 1] for further details.

Data Cache Analysis

The analysis of data caches is very similar to that for instruction caches. The additional part consists in performing a data flow analysis to discover the addresses accessed by `load` and `store` instructions and to determine which cache blocks are used [8].

While an instruction fetch addresses a single memory block (known at analysis time), an access to data performed within a loop may reference several memory blocks across iterations. The data flow analysis may find that the target of a memory access has a single value, or a set or an interval of values. A special value, $\top$, represents the cases where the analysis fails to determine the possible address values. Such a failure could reflect input-data dependent addresses or might be due to an imprecise address analysis that makes over-approximations.

The imprecision in the address analysis is also a source of imprecision in the computation of the abstract cache states used to assign the categories. When a single address value has been found for a load/store instruction, the cache state is updated in the same way as for an instruction cache. If the instruction may reference several different addresses, the analysis accounts for a possible impact on several cache sets (on all cache sets if the predicted address is $\top$. For the MUST analysis, all blocks are aged by one (whatever the number of possible address values, only one is accessed at a time); in the MAY analysis, the age of all accessed blocks is unchanged while other blocks are aged. In the end, an imprecision in the address analysis directly induces over/underestimation (MUST/ MAY) of ages in the cache states and hence on the categories: memory access with imprecise address sets are more likely to be categorized as *NC*.

Exploitation of Cache Categories to Compute the WCET

A straightforward way to account for the cost of instruction and data cache misses in the estimated WCET of a program is to add it to the expression of the execution time so that the ILP solver maximises the number of misses together with the execution time. However, this solution is reliable only if the execution time of a sequence of instructions experiencing a cache miss is less than the adding the the cost of that miss to the execution time with a hit. This is not the case when the program runs on an architecture that enables timing anomalies [11], i.e. for which the local worst case for the execution time of the sequence might be a hit.

A more reliable approach is to compute the local WCET of each basic block in the program CFG as many times as possible combinations of hits and misses for *NC* accesses exist. Then the execution count of each of these different WCET values is bounded by a combination of category-related execution counts in the ILP formulation. This approach has been implemented in OTAWA [2], the static WCET analysis tool used to carry out the experiments reported in this paper. In OTAWA, a local WCET is computed for each sequence of two basic blocks (then related to an edge in the CFG) based on contextual execution graphs [13]. It considers an optimistic or pessimistic context (availability of hardware resources) for the first and second block in the sequence, respectively, thus ensuring that an upper bound is found.

3.2 Contribution to the Lower Bound on the WCET (WCET$^-$)

Approximation in category-based instruction cache analysis clearly comes from the join operator $\mathcal{J}^{\#}$: even if the input cache states are precise, the result may be imprecise. In data cache analysis, another source of approximation is the address analysis that sometimes fails to derive precise address values, which leads to additional inaccuracy in the computation of

Table 1 Benchmarks.

benchmark	# accesses to I$	# loads
cjpeg_jpeg6b_wrbmp	37,328	6,187
gsm	4,2319,41	895,605
gsm_decode	1,739,854	211,854
gsm_encode	2,378,890	681,917
h264dec_ldecode_block	11,942	1,802
h264dec_ldecode_macroblock	51,373	51,373

cache states. Imprecise cache states lead to imprecise categories (*NC*, and *PERS* to a lesser extent) then to possibly overestimating WCET^+.

A simplistic approach to estimate WCET^- would consist in considering each *NC* access to the cache as a hit (similarly, each *PERS* access as a hit in the first loop iteration). However, to account for possible timing anomalies, both options must still be considered (as explained in Section 3.1). More precisely, an opposite assumption is taken for each basic block in a sequence. For example, all *NC* accesses in the first block are accounted for as misses, while those in the second block are considered as hits. This way, the local WCET of the second block is minimized (assuming no timing anomaly can occur). At the end, when all possible scenarii have been explored, the lowest local WCET is kept for each basic block.

4 Experimental Study and Discussions

4.1 Methodology

All the experiments reported here have been done using our OTAWA toolset [2] in which we implemented support for the analysis of the possible overestimation.

We considered a microprocessor architecture with a 7-stage in-order pipeline similar to that of the MPC5554 from Freescale. We assumed a single level of separated instruction and data caches with the following parameters: 4KB of size, 4-way associative with LRU replacement, with 16B cache lines and a write-allocate policy. A memory latency (for cache misses) of 20 cycles was considered. We considered an infinite-capacity write buffer, then only loads are considered in the experiments.

We have analysed several benchmarks from the MediaBench suite [9]: cjpeg_jpeg6b_wrbmp is a JPEG image bitmap writing code; gsm, gsm_decode and gsm_encode are respectively a GSM 06.10 provisional standard codec, decoder and encoder; h264dec_ldecode_block and h264dec_ldecode_macroblock are H.264 block and macroblock decoding functions. The dynamic numbers of accesses to the instruction cache and loads of these benchmarks (i.e. counts on the worst-case path) are given in Table 1.

We assume that each execution path that fulfils the flow facts specified to the WCET analysis tool and converted into constraints in the ILP formulation is feasible (i.e. no overestimation comes from the flow analysis). Similarly, we assume that any initial hardware state is feasible.

Table 2 Possible overestimation on WCET^+.

benchmark	$U = (\mathsf{WCET}^+ - \mathsf{WCET}^-)/\mathsf{WCET}^-$
cjpeg_jpeg6b_wrbmp	36.25%
gsm	161.95%
gsm_decode	96.33%
gsm_encode	216.00%
h264dec_ldecode_block	126.11%
h264dec_ldecode_macroblock	144.41%

Table 3 Possible overestimation in instruction cache analysis.

benchmark	*PERS*	*NC*	$U_{i\$}$
cjpeg_jpeg6b_wrbmp	25.0%	5.1%	0.54%
gsm	34.9%	19.6%	2.36%
gsm_decode	38.4%	29.3%	1.83%
gsm_encode	31.5%	18.7%	2.29%
h264dec_ldecode_block	52.6%	18.8%	4.84%
h264dec_ldecode_macroblock	57.7%	1.8%	0.32%

4.2 Results

Table 2 shows the possible overestimation on (WCET^+). For cjpeg_jpeg6b_wrbmp), the maximum overestimation on WCET^+ is small (36.25%), meaning that the analysis is precise in this case. The other benchmarks exhibit much larger potential overestimation, ranging from 96.33% to 216%. As we will see below, this is mainly due to the weakness of the data cache analysis we considered, which fails to compute the targets of load instructions.

In the following, we try to isolate the respective contributions of the instruction and data cache analyses to the possible overestimation of the WCET upper bound. For that purpose, we compute $U_{i\$}$ (resp. $U_{d\$}$) considering pessimistic results for the other analysis.

Table 3 shows the contribution to the overestimation by the instruction cache analysis ($U_{i\$}$). These low numbers (less than 5%) show how instruction caches can be taken into account with much accuracy in WCET analysis. There are two reasons for this. First, the instruction cache analysis generates relatively precise results: columns 2 and 3 show how many accesses are classified as *PERS* (*Persistent*, generating a hit for all loop iterations but the first one) or *NC* (*Not Classified*, then considered as a hit to be optimistic and as a miss to be pessimistic). Except for one benchmark, the number of instruction fetches that cannot be classified is less than 20%. Second, the impact of a miss on an instruction fetch can sometimes be partly hidden by stalls generated by instructions inter-dependencies, and this is captured by our pipeline analysis.

The major part of the possible overestimation on the static WCET comes from the data cache analysis, as shown in Table 4. Contrarily to the instruction cache, many accesses (from 46% to more than 80%) are categorized as *NC*. This is due to the basic address analysis implemented in OTAWA, that only handles global and stack data. Addresses of accesses to array elements are not determined by this analysis. As reported in column 4, the precise

Table 4 Possible overestimation in data cache analysis.

benchmark	*PERS*	*NC*	*unknown @*	$U_{d\$}$
cjpeg_jpeg6b_wrbmp	9.2%	46.1%	33.8%	35.29%
gsm	0.0%	55.8%	38.4%	66.25%
gsm_decode	0.0%	69.4%	52.6%	143.66%
gsm_encode	0.0%	51.7%	34.1%	190.01%
h264dec_ldecode_block	0.0%	80.5%	64.4%	82.51%
h264dec_ldecode_macroblock	0.0%	54.4%	22.1%	141.90%

target address could not be determined by the data analysis for a considerable amount of load instructions (from 22% up to 64%). An unknown target address impacts the load instruction itself, since it cannot be determined whether it hits or misses in the cache, but it also impacts subsequent loads by producing a destructing effect on the abstract cache state. Clearly, the address analysis can be identified as the weakest link here.

4.3 Possible Interpretations of the Lower Bound on the WCET

Several uses of the overestimation metric (U) – difference between the upper and lower bounds on the WCET – can be envisioned:

- An obvious application is the qualification of a particular estimated WCET. If U is small, then the estimated WCET can be considered as precise. This means that WCET^+ is not far above the real WCET (even if measured execution times look much lower). Then the estimated WCET will not drive to oversize the hardware required to meet deadlines, nor break irrelevantly the real-time constraints verification. Instead, if U is large, this may mean either that the WCET computation approach does not fit the particular profile of application under analysis, or that the way the application is coded drives too many non predictable behaviours.
- Local contributions to overestimation could also be used to implement an adaptive WCET analysis. For example, a first fast analysis could be performed with a very simple (but imprecise) method. Then the program parts exhibiting the largest overestimation could be processed again with a more precise but also more time-consuming method. Abstract interpretation is well adapted to support such an approach: to enhance accuracy, the analysis could choose which states would be merged and which would be kept separate by the *Join* operator.
- Finally, the overestimation metric could also be considered as a basis to compare different analyses with respect to the precision they can achieve.

5 Conclusion

Static WCET analysis has the reputation of overestimating WCET, which is not completely false since it is designed to compute upper bounds on the real WCET. However the size of the gap between the real and estimated WCETs is unknown (since the real WCET cannot usually be determined). Comparison of static WCET estimates to measured execution times often shows large discrepancies but nobody can tell whether the upper bound is too pessimistic or if the measured execution time was observed under particularly favourable conditions.

Our objective in this paper was to propose a framework to evaluate the possible overestimation of static WCETs. We defined a *lower bound on the real WCET* (WCET^-). This value can been seen as a complement to the traditional upper bound, WCET^+. It represents an execution time that *can* be reached, provided all information on infeasible paths and initial hardware states were known and taken into account at analysis time. The real WCET is guaranteed to be between the lower and upper bounds. We envision that these information could be useful not only to increase confidence in the static WCET analysis but also to guide code optimisations to enhance precision. It could also be used to compare analysis techniques. Note that any observed execution time that would be greater than WCET^- could be used to tighten the interval in which the real WCET falls, and thus to improve precision.

As future work, we plan to apply the proposed approach to other hardware components in single- and multi-core architectures. We will also investigate how it can be used to evaluate the precision of flow analysis techniques.

References

1 Clément Ballabriga and Hugues Cassé. Improving the first-miss computation in set-associative instruction caches. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, 2008.

2 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, 2010.

3 Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2), 2000.

4 Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

5 Niklas Holsti et al. WCET Tool Challenge 2008: report. In *Workshop on Worst-Case Execution Time Analysis*, 2008.

6 Christian Ferdinand. A fast and efficient cache persistence analysis. Technical report, Saarländische Universität (Germany), 2005.

7 Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying compiler techniques to cache behavior prediction. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1997.

8 Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Languages, Compilers, and Tools for Embedded Systems*, 1998.

9 Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Intl symposium on Microarchitecture*. IEEE Computer Society, 1997.

10 Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modelling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium (RTSS)*, 1995.

11 Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Real-time systems symposium (RTSS)*, 1999.

12 Fadia Nemer, Hugues Cassé, Pascal Sainrat, and Jean-Paul Bahsoun. Inter-Task WCET computation for A-way Instruction Caches. In *Symp. on Industrial Embedded Systems (SIES)*, 2008.

13 Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. 2009.

14 V. Suhendra, T. Mitra, A. Roychoudhury, and Ting Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Design Automation Conference (DAC)*, 2006.