WE-HML: hybrid WCET estimation using machine learning for architectures with caches

Abderaouf Nassim Amalou, Isabelle Puaut, Gilles Muller

▶ To cite this version:

Abderaouf Nassim Amalou, Isabelle Puaut, Gilles Muller. WE-HML: hybrid WCET estimation using machine learning for architectures with caches. RTCSA 2021 - 27th IEEE International Conference on Embedded Real-Time Computing Systems and Applications, Aug 2021, Online Virtual Conference, France. pp.1-10. hal-03280177

HAL Id: hal-03280177
https://inria.hal.science/hal-03280177

Submitted on 7 Jul 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

WE-HML: hybrid WCET estimation using machine learning for architectures with caches

Abderaouf N Amalou
Univ. Rennes
Rennes, France
abderaouf.amalou@irisa.fr

Isabelle Puaut
Univ. Rennes
Rennes, France
isabelle.puaut@irisa.fr

Gilles Muller
Inria, Paris
Rennes, France
Gilles.Muller@inria.fr

***Abstract*—Modern processors raise a challenge for WCET estimation, since detailed knowledge of the processor micro-architecture is not available. This paper proposes a novel *hybrid* WCET estimation technique, WE-HML, in which the longest path is estimated using static techniques, whereas machine learning (ML) is used to determine the WCET of basic blocks. In contrast to existing literature using ML techniques for WCET estimation, WE-HML (i) operates on binary code for improved precision of learning, as compared to the related techniques operating at source code or intermediate code level; (ii) trains the ML algorithms on a large set of automatically generated programs for improved quality of learning; (iii) proposes a technique to take into account data caches. Experiments on an ARM Cortex-A53 processor show that for all benchmarks, WCET estimates obtained by WE-HML are larger than all possible execution times. Moreover, the cache modeling technique of WE-HML allows an improvement of 65 percent on average of WCET estimates compared to its cache-agnostic equivalent.**

I. Introduction

The Worst-Case Execution Time (WCET) of a program is the longest time the program will take to execute on a given architecture. Knowing the WCET of a program is crucial in real-time systems to prove that deadlines will be met. Determining the exact WCET of a program is not tractable, the WCET is therefore *estimated*, seeking for an upper bound of the exact WCET. The challenge addressed in this paper is to compute WCET estimates for modern processors, for which the processor's micro-architecture is not precisely known.

WCET estimation methods are divided between *static* methods, *end-to-end measurement-based* methods and *hybrid* methods [1]. Static methods estimate the WCET without executing the program. In the first phase, they estimate the WCET of each basic block in the program thanks to the knowledge of the architecture, and in the second phase they calculate the whole program's WCET estimate from those of the basic blocks. For the second phase, *Implicit Path Enumeration Technique* (IPET) [1] is the most used class of techniques. IPET relies on solving a linear optimization problem generated from the program's Control Flow Graph (CFG). Static methods provide a *safe* WCET estimate, which is an upper bound of any possible execution time, provided that the WCET estimate of each basic block is itself safe. However, static methods require extensive knowledge of the micro-architecture of the processor (caches [2], pipelines [3], branch predictors [4]). This information is increasingly difficult to obtain with recent architectures, for intellectual property reasons, or simply because the micro-architecture is too complex to design a safe and accurate model of the processor's timing.

End-to-end measurement-based methods are empirical techniques that do not require detailed knowledge of the hardware. They launch the program on a series of inputs, and the resulting execution times are measured and gathered. The WCET is then estimated, either by taking the highest measurement, or by extrapolation using statistical techniques (Measurement-Based Probabilistic Timing Analysis, or MBPTA [5]). By construction, when using the highest measurement as WCET estimate, these techniques can only underestimate the WCET, unless the input and the hardware state resulting in the longest execution path are used during the tests [6]. Therefore, a safety margin is often added to the WCET estimate to mitigate the lack of confidence in the measurements.

Hybrid methods mix static and measurement-based approaches. In a vast majority of these techniques (e.g. [5], [7]–[10]), *measurements* are used to estimate the WCET of basic blocks. The WCET of the whole program is then estimated using calculation methods such as IPET. The advantage of hybrid techniques is that they do not require knowledge of the architecture, while being able to find the longest path of the CFG. However, measurement-based hybrid methods are prone to the problem of measuring the execution time of every basic block b at least once (*code coverage* issue) [11]: finding an input for which b is executed is challenging, as well as finding an input that exercises the worst-case execution scenario of b.

In this paper we propose a new hybrid WCET estimation technique named WE-HML (for **W**CET **E**stimation using an **H**ybrid **M**achine-**L**earning based technique) that supports caches. WE-HML operates in two phases. In the first phase (*learning phase*), the timing model of the processor is learnt through the training of machine learning (ML) algorithms. WE-HML comes with five ML algorithms that are found to produce good timing predictions. Learning is performed using extensive measurements on the binary code of a large set of automatically generated basic blocks. The WCET of each basic block is learned for different *execution contexts* of the basic block. For the scope of this paper, the considered context for a basic block is the level of *cache pollution* coming from the execution of other code in the same loop nest. One benefit of WE-HML is

that it requires little knowledge on the memory hierarchy. The training phase is executed only once per architecture.

In the second phase (*WCET estimation phase*), the WCET of each basic block of the target program is computed, by applying the timing model learnt in the first phase for the cache pollution suffered by the basic block. The cache pollution is calculated using static analysis. The WCET of the overall program can finally be computed using a modified version of IPET. This second phase is executed once for each target program.

We believe that WE-HML can be used for software at intermediate safety levels (for example DAL B and C in the aeronautic industry [12]), that may use for obvious cost reasons, processors that are too complex to have a safe timing model, and thus for which static WCET estimation methods are not available. For such systems, WCETs are needed, and thus some pessimism in WCET estimation is tolerated, but missing a deadline, if sufficiently rare, can be accepted.

We have evaluated WE-HML on the Cortex-A53 processor used in the Raspberry Pi 3 B+ platform [13], for which there is no detailed micro-architecture description available at the current time. Our experiments evaluate the WCET of 13 programs from the TACLeBench benchmark suite [14]. Since WE-HML is an empirical technique (the learning phase uses measurements), there is no formal guarantee that it produces safe WCET estimates. Still, for the tested programs, estimated WCETs are always larger than the maximum observed execution times (MOETs).

WE-HML has two advantages over existing hybrid techniques. First, measurements, which are time-consuming, are performed only once per architecture, to train the ML algorithms. Estimating the WCET of a target program is fast based on our experimental evaluation, about thirty seconds for most programs. Second, since the WCETs of the basic blocks of a program are estimated using an ML algorithm instead of measurements, WE-HML eliminates the code coverage issue that exists in related hybrid techniques.

While machine learning has been used in the past to estimate WCETs [15]–[17], all methods that we are aware of, operate at source code or intermediate code level. WE-HML instead operates on *binary code* to augment the precision of learning. In addition, most ML-based WCET estimation techniques apply training on a small number of benchmarks, whereas WE-HML relies on a large set of automatically generated basic blocks, providing a high quality of training. Finally, WE-HML is unique in its support for caches. The contributions of this paper are the following:

- A hybrid WCET estimation technique for single-core processors, based on an ML-derived timing model of the core.
- A technique to account for processor caches, with little knowledge of the memory hierarchy.
- A selection of five ML algorithms, based on the learning scores for basic blocks obtained after experiments (*r2* score [18], mean relative error and maximum error): Random Forests, Neural Networks, Gradient Boosting, Ridge and Bayesian Ridge. None of these five techniques consistently outperforms the others for all benchmarks.
- A detailed experimental evaluation of the quality of predictions and the interest of accounting for caches.
  - When evaluating WE-HML on complete programs, we observe that predicted WCETs are always higher than any observed execution times, for all benchmarks and all ML algorithms.
  - By comparing WE-HML (that accounts for cache effects) with a cache-agnostic technique, we observe that cache modeling decreases WCET estimates by 65% on average.
  - Finally, our experimental evaluation provides a comparison with a cache-agnostic measurement-based hybrid approach. Experimental results show, on a representative benchmark, that WE-HML calculates WCET estimates that are 2.5 times smaller than the baseline hybrid technique.

For the sake of reproducibility of results, the source code of WE-HML is available (https://gitlab.inria.fr/puaut/we-hml/).

WE-HML currently targets single-core processors. Extending our work to multi-core processors will require taking into account interference delays when accessing shared hardware resources, which is left for future work.

The rest of this paper is organized as follows. Section II presents the WE-HML approach. The experimental methodology for evaluating WE-HML on an ARM Cortex-A53 processor is detailed in Section III. Experimental results are given in Section IV. Section V compares our approach to related techniques. We finally discuss on the results achieved and present our future work in Section VI.

II. WE-HML APPROACH

WE-HML operates in two phases. In the first phase, described in Section II-A, machine learning algorithms are trained, using measurements on a large set of automatically generated basic blocks. In a second phase, presented in Section II-B, the WCET of programs is estimated using a modified IPET calculation method. For the sake of clarity, these two phases are presented in a target-independent manner, and voluntarily disregard the consideration of the execution context of basic blocks (for the scope of this paper, caches), which are detailed in Section II-D. Automatic generation of training data is presented in Section II-C.

A. Learning the processor timing model (training)

The training phase is executed once per target architecture. Its purpose is to learn the timing model of the processor, as depicted in Figure 1. WE-HML comes with five ML algorithms, that are trained on automatically generated basic blocks. The automatic generation of basic blocks, detailed in Section II-C aims at covering a large variety of code structures that exist in real code. Once trained, each ML algorithm can estimate the WCET of any basic block in programs, including basic blocks never encountered during the training phase. The ML algorithm captures the impact of the contents of a generated basic block on its WCET.

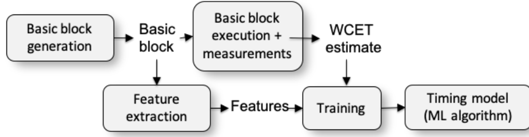

Fig. 1. WE-HML training phase

The ML algorithm learns from the values of numerical quantities, called *features*. The considered features in WE-HML are a vector of proportions of each type of machine instruction (e.g. add, sub) to the number of instructions in the considered basic block ($\frac{\#specific_instr}{\#instrs}$). When an instruction type has different addressing modes that impact the instruction timing (i.e., memory vs register operands), each variant is a different entry in the vector. Encoding instruction types as proportions permits the construction of a timing model that is independent of the length of basic blocks. For the same reason, the WCET estimate of a basic block is also encoded as a proportion of cycles to the number of instructions in the basic block ($\frac{WCET}{\#instrs}$). Features and normalized WCET estimates are both represented as floating-point values.

B. Estimating the WCET of a target program

The WCET estimation phase for a target program is shown in Figure 2. First, basic blocks, their associated features, and the program's Control Flow Graph (CFG) are extracted from the program's binary code. The learned timing model is then used to compute a WCET estimate for each basic block. The CFG and the WCET estimates are then fed back to a WCET estimation tool that implements the IPET [19] for estimating the WCET of the entire program. In our WE-HML prototype, we have modified the IPET implementation of the Heptane open-source software [20].

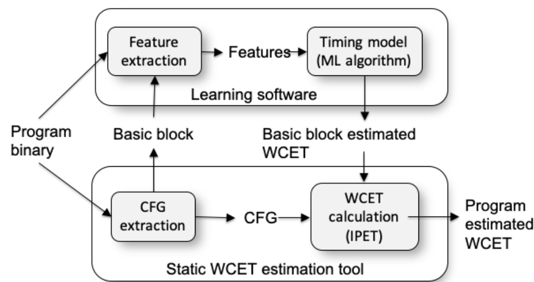

Fig. 2. WE-HML WCET estimation phase

Listing 1. Example of C code

```
for (int i = 0; i < 100; i++)
    if (t[i]>0) s = s + t[i];
    else s = s - t[i];
```

We illustrate WCET estimation on a simple program that computes the sum of the absolute value of 100 elements stored in an array t. Listing 1 and Figure 3 show respectively the C source code and the corresponding CFG that is extracted from the compiled binary code. In Figure 3, nodes correspond to basic blocks and edges correspond to possible control flow between them. For example, basic block *then* contains the code for *s=s+t[i]*.

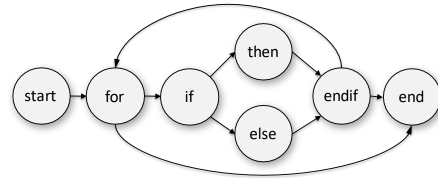

Fig. 3. Control flow graph for code of Listing 1

$$\begin{aligned} n_{start} &= 1 \\ n_{for} &\leq 101 \\ n_{for} &= n_{start\to for} + n_{endif\to for} \\ n_{for} &= n_{for\to if} + n_{for\to end} \\ n_{if} &= n_{for\to if} \\ n_{if} &= n_{if\to then} + n_{if\to else} \end{aligned}$$

Fig. 4. IPET formulas for CFG of Figure 3

First, the WCET w_b of a basic block b is estimated by applying the ML algorithm. Then, the IPET technique estimates the longest path in the program using integer linear programming: the goal is to maximize the following quantity: $\sum_{b\in \text{basic blocks}} w_b \times n_b$, with n_b the number of executions of basic block b.

Constraints on variables n_b and $n_{b\to b'}$ (the number of times the edge $b \to b'$ is taken, b and b' being basic blocks) model the execution flows (a basic block is entered as many times as it is exited) and the maximum numbers of iterations for loops. Constraints are generated by the IPET technique, possibly with annotations for loop bounds when the tool is not able to infer them automatically. An excerpt of the constraints for the example program is given in Figure 4. Assuming for the sake of illustration that the outcome of the learnt timing model is a WCET of 10 cycles for all basic blocks, except block *then* which executes in 20 cycles, the result of the IPET calculation for the example is then $n_{start} = 1, n_{for} = 101, n_{if} = 100, n_{then} = 100, n_{else} = 0, n_{endif} = 100, n_{start} = 1$ and the WCET estimate is 5030.

For simplicity reasons, we have assumed for this illustrative example that each basic block has a single, context-independent WCET estimate. For architectures with caches, this assumption is obviously not valid anymore. In Section II-D, we show how to extend this simple formulation to take caches into account.

C. Automatic generation of training data

Existing works using machine learning algorithms for WCET estimation [15], [16] rely on a small number of benchmarks to train the ML algorithms. This may limit the quality of training, because the amount of training data is too small, and the code snippets may be too homogeneous. Similar to [21], we address this issue by using a large set of automatically generated basic blocks as training data. The WE-HML code generator relies on a grammar to generate source code (C code) for basic blocks, that is subsequently compiled into binary.

The basic blocks produced by the WE-HML code generator have randomly selected numbers of statements and variables. The generated code uses all the standard basic types, all selected randomly by the generator, with user-provided parameters specifying the proportion of each type: *char*, *short*, *int*, *long*, in their signed and unsigned variants, as well as arrays of basic types. The most common C operations (arithmetic and logical operations, array indexing, shift and rotate operations, binary and unary operators on booleans, etc.) are covered.

A generated basic block first declares a set of variables and then applies randomly-selected operations on these variables. The code is guaranteed by construction to not trigger any exception at run-time (e.g. no out-of-bound array accesses). In order to cover branch instructions, the code may contain *if* statements. However, the generator ensures that there is no data-dependent execution, and makes the outcome of conditional branches always known, i.e., the condition of the test for *if* statements is always *true*. The definition of a basic block in WE-HML is not the classical definition of a basic blocks as defined in the compiler domain. A basic block in WE-HML may contain branches, for the sake of timing estimation of branch instructions. An example of a generated basic block is given in Listing 2.

Listing 2. Example of generated basic block

```
array_0[233] = ( one > zero ) ? var_5 : var_4;
var_3 = array_0[array_index] - var_3;
array_0[164] = var_3 << small_int;
if (one > zero) {
  var_1 = var_5 % 65497;
  var_6 = var_6 >> 2;
  array_0[146] = -array_0[array_index]; }
```

Since the WE-HML code generator produces C code, it is not dedicated to a particular architecture, and thus it can be used for WCET estimation of other targets than the ARM Cortex-A53 used in this paper.

The experimental conditions used to avoid as much as possible bias when training the ML algorithms on automatically generated code (e.g. optimistic timing for branches) will be detailed in Section III.

D. Supporting processors with caches

The memory hierarchy has a significant impact on the execution time of a basic block. When the instruction/data caches contain no information (*cold* cache), or worse, when dirty data has to be copied-back in memory, the execution time of a basic block is much longer than when the cache contains useful information loaded previously (*warm* cache). Therefore, not considering the memory hierarchy during WCET estimation amounts to evaluating only the cold cache scenario, which may result in highly pessimistic WCET estimates.

In contrast to static cache analysis techniques that require precise knowledge of both the cache architecture and the memory accesses made by the target program, WE-HML takes cache hierarchies into account by *learning* the impact of caches on the WCET of basic blocks.

A basic block executed within a loop nest may experience different levels of pollution of the cache hierarchy depending on the other memory accesses performed in the same loop nest. This is captured by the concept of cache *pollution value*. For a given basic block b executed repetitively within a loop nest, the pollution value models the volume of data that may conflict in the cache between successive executions of b in the loop nest. More precisely, if basic block b accesses x bytes of data, a *pollution value* of p means that $p * x$ bytes are accessed within the loop nest outside b, and may evict the data loaded by b. We observed that the larger the pollution value, the higher the execution time of the basic block.

As part of the training phase, learning is performed for different pollution values. The WCET of each basic block b for a pollution value p is estimated by wrapping the code of b in a function, whose code is given in Listing 3.

Listing 3. Executing a basic block with cache pollution

```
flush_caches();
for (i = 0; i < nb_iter; i++) {
  // Monitor exec. of BB (read cycle counter)
  cnt_read(&tb); BB(); cnt_read(&ta);
  invalidate_icache();
  pollute(p*x); // Write randomly p*x bytes
}
```

The basic block is executed multiple times and its execution time is measured by reading the processor cycle counter (call to function *cnt_read*). *pollution code* is inserted between the successive executions (function *pollute*). The objective of the pollution code is to evaluate the performance loss resulting in the execution of other basic blocks in the same loop nest, caused by pollution of all the caches in the memory hierarchy. The timing of the first execution, corresponding to the *cold cache scenario* is discarded, and the WCET for the warm cache scenario is evaluated from the measurements (see details in Section III-C). In order to minimize the amount of knowledge on the loop nest inside which the basic block is executed, the pollution code experimentally explores the memory references from the enclosing loop nest: for a pollution value p, the pollution code randomly accesses $p * x$ bytes in an large array (whose size is the size of the last-level cache) to find the worst case. The repetitive execution of the basic block serves two purposes: (i) capturing the inherent timing variability of the processor; (ii) capturing (with no formal guarantee) the worst-case pollution from other blocks in loop nests.

The pollution code allows learning the impact of *all* cache levels in the memory hierarchy on the execution time of the basic block under study. This is achieved without precise knowledge of the different cache levels; WE-HML only requires the knowledge of the size of the last level cache, which can be easily determined experimentally. To be on the safe side, since pollution in the instruction cache is not yet managed, the instruction cache is flushed between experiments (call to *invalidate_icache*).

As part of the estimation phase, WE-HML first estimates for each basic block executing within a loop nest the cache pollution value p. This is done using a simple static program analysis (see details of the implementation in Section III-D) that counts the volume of data accessed within the loop nest.

Two WCET values per basic block are then used during WCET estimation: one value for the first execution of the basic block (*cold cache* scenario), obtained by using the ML prediction for the largest pollution value, and one value for the next executions of the basic block within the loop nest (*warm cache* scenario), obtained by using the ML prediction for the pollution value predicted statically. These two values are fed to the IPET calculation technique, with an extra constraint in the IPET calculation technique constraining the *cold cache* value to be used only for the first iteration of the loop.

III. Experimental setup

In this Section, we detail the experimental setup used for the evaluation of WE-HML for the Raspberry Pi 3 B+ platform. The hardware and software environments are first introduced (Section III-A). The programs used for evaluating the quality of predictions are presented (Section III-B). We then detail the learning and prediction phases of WE-HML (Sections III-C and III-D).

A. Hardware and software environments

The *Raspberry Pi 3 B+* [13] relies on a Broadcom BCM2837 SoC which is based on a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor, a 2-wide superscalar processor. The architecture features a private L1 cache and a 512 KiB shared L2 cache. Timing measurements are obtained using the cycle counter implemented in the processor (function *cnt_read* in Listing 3). Reading the cycle counter requires one machine instruction, resulting in negligible measurement overhead.

The Raspberry Pi runs the Raspbian Lite operating system (Linux kernel version 4.19, light operating system without user interface to minimize the impact of the operating-system activity on timing). Similar to Bate et al. in [22], we configure the Raspbian operating system to force a constant processor frequency (800 MHz) and thus avoid Dynamic Voltage Scaling (DVS). To avoid as much as possible timing noise coming from the operating system, the codes are compiled and run as Linux kernel modules. The compiled codes are executed on a specific core (core 3) on which no user task is allowed to run (isolated core, using the Linux *isolcpus* facility). A cold cache is enforced at the beginning of each experiment, by invalidating the instruction cache and filling the data cache with dirty data.

B. Benchmarks

The quality of WCET predictions of programs was evaluated on 13 benchmarks from the TACLeBench benchmark suite [14]. Benchmarks using floating-point numbers were discarded because execution in kernel mode does not support floating-point values. We also excluded the benchmarks using emulated instructions, and the benchmarks reaching the limits of the prototype (using recursion or having complex call graphs not yet supported by cache pollution computation, as detailed in Section III-D). Table I gives the main characteristics of each benchmark: a brief description, the maximum depth of loop nesting found in the code, and the number of basic blocks.

TABLE I
PROPERTIES OF BENCHMARKS

Name	Description	Nest.	#BB
binarysearch	Binary search in an array	1	24
bsort	Bubble sort algorithm	2	33
countnegative	Basic counting on arrays	2	34
crc	Cyclic redundancy codes	1	30
expint	Exponential integral function	2	30
fdct	Fast discrete cosine transform.	1	10
fir	Finite impulse response filter	2	16
h264_dec	H.264 block decoding functions	5	165
insertsort	Insertion sort	2	10
jfdctint	Discrete-cosine transformation	1	12
matrix1	Generic matrix multiplication	3	35
ns	Search in 4-dimension array	4	19
petrinet	Petri net simulation	1	170

Each benchmark comes with input values exercising the longest execution path.

Compiler optimizations were disabled when compiling the benchmarks, to facilitate the provision of flow information during WCET analysis (if optimizations were allowed, flow information, for example, loop bounds would then have to be transformed manually according to the optimizations applied by the compiler, which is error-prone [23]–[25]). Consideration of compiler-optimized code is left for future work.

The original code of benchmarks *expint* and *ns* was containing a long piece of code executed in only one loop iteration. As Heptane, the tool we have modified does not include any detection of such an *infeasible* path [26], it considers that this path is executed at all iterations, resulting in highly overestimated WCETs. Since we aim at estimating the quality of WE-HML and not the quality of Heptane, the code of these two benchmarks was re-structured to avoid this infeasible path.

C. Training phase

A total of 15000 basic blocks was generated for the training of the ML algorithmes. We have made sure that the generated basic blocks cover all the instructions that are produced by *gcc* with no optimization for the ARM Cortex-A53 processor. Because the list of instructions possibly generated by *gcc* is not documented, we have checked that all instructions used in the benchmarks are present in the generated basic blocks. The training phase was performed on for 10 pollution values, which are powers of two ranging from 1 to 512, resulting in 10 variants of each tested ML algorithm, one per pollution value. The highest pollution value of 512 was experimentally determined by analyzing the impact of pollution on a large number of basic blocks.

The quality of the timing model depends on the input training data, which has to be as representative as possible of the real world it is intended to represent. A lot of attention was paid to avoid biased training data as much as possible:

- As the timing of basic blocks mainly depends on the type of instructions executed, the parameters of the code generator were tuned to have proportions of instructions similar to real code while avoiding being too close to any

particular benchmark. We also tuned the parameters to cover varied sizes of basic blocks, from very small basic blocks to longer ones.

- The execution time of specific instructions should not be smaller during the training phase than during a real execution. This could occur for branches, for which the repetitive execution of basic blocks used to collect training data could introduce a bias. This bias was eliminated by flushing the branch prediction tables between each timing measurement in the measurement loop from Listing 3. Regarding the effect of caches, the presence of pollution code aims at maximizing the execution time of a basic block in presence of cache pollution within loops.
- A foreseen bias in training data is that we only consider *proportions* of instructions during training, regardless of their *order of execution* in the basic block. This choice was taken to have fast training and prediction.

We implemented two variants to estimate the WCET of basic blocks from the set of collected measurements:

- The WCET estimate is set to the largest observed execution time (MOET). For this technique, each basic block is executed a sufficiently large number of times (1000 in our experiments) to cover at best the possible timings.
- The WCET estimate is estimated using Measurement-Based Probabilistic Timing analysis [5], [27], with an exceedance probability of 10^{-3}. Extreme value theory (EVT), and more precisely GEV (Generalized Extreme Value theory) was used to obtain the WCET from a set of measurements. We made sure that the timing samples respect the three conditions that must hold to apply EVT: stationery, short-term independence, and long-term independence [28]. 200 runs were sufficient to make sure that 10000 basic blocks, out of 15000 generated basic blocks, respect the three applicability conditions [29] with a commonly-used significance level of 5%.

For both techniques, 80% of the basic blocks were used for training and cross-validation, 20% were used for testing.

Executing the 15000 basic blocks to obtain the timing samples for the 10 pollution values required approximately 36 hours, using a single Raspberry Pi 3B+ board. We do not consider the duration of the training phase to be an issue, since it has to be performed only once per architecture and could be easily executed in parallel on several boards. Training, executed on a Linux virtual machine running on a DELL Latitude 7400 with 8 core Intel i7 processor, required around 62 minutes in total for the 5 ML algorithms.

TABLE II
EXPERIMENTED MACHINE LEARNING ALGORITHMS

Algorithm	Description
Random Forest (RF)	A multitude of decision trees
Neural Network (NN)	Multi-Layer Perceptron neural network
Gradient Boosting (GB)	Stochastic gradient boosting
Bayesian Ridge (BR)	Bayesian ridge regression
Ridge	Standard ridge regression

We have evaluated the machine learning algorithms that provided by the Scikit library [18], [30]. Preliminary experiments made us select the 5 algorithms that gave the best results among those provided by the Scikit library (see Table II for a brief description). In the rest of the paper, the acronyms (*RF*, *NN*, *GB*, *Ridge*, *BR*) will be used instead of the full names.

D. WCET estimation phase

WCET estimation is implemented by modifying the open-source IPET-based static WCET estimation tool Heptane [20] as follows. Heptane was modified to calculate the pollution value for each basic block. The current estimation of the pollution value is conservative, in the sense that it considers *all* accesses inside a loop nest: in case there are multiple paths in a loop, the number of accesses in the different paths are summed, leading to safe but overestimated pollution values. Another source of conservatism is that the presence of loops is considered as the only source of cache reuse: reuse resulting from function calls is currently ignored. In the current state of the implementation, cache pollution is computed only for loop nests that contain a function call tree of depth strictly higher than one, and the benchmarks that do not meet this condition are discarded.

Then, the IPET calculation step of Heptane is modified to use the ML-predicted WCET values for basic blocks instead of the values predicted by static analysis as in the original Heptane. Two WCET estimates for each basic block are predicted: one estimate with a cold cache (for the execution of the basic block within the first loop iteration), and a second with a warm cache, using the statically-predicted cache pollution value. The original calculation step of the Heptane is then applied, using these two WCET estimates.

IV. EXPERIMENTAL RESULTS

The quality of WE-HML is evaluated from different points of view. First, we evaluate the quality of WCET predictions of entire programs (Section IV-A). Then, we evaluate the benefit of accounting for caches (Section IV-B). WE-HML is then compared with a cache-agnostic measurement-based hybrid technique in Section IV-C. Finally, a detailed analysis of the quality of WCET predictions at the basic block level is given in Section IV-D.

A. Prediction of WCETs of programs

Table III reports the WCET estimated by WE-HML (with cache modeling) on the benchmarks, using the 5 selected ML algorithms. The estimated WCETs are compared with the maximum observed execution time (MOET) of each benchmark, obtained by taking the maximum timing of 1000 executions, all using the inputs that trigger the worst-case execution path. The predicted WCET values in the Table are obtained by the best-performing variant of WE-HML in terms of quality of learning for basic blocks: training with pWCET-10^{-3} values for basic blocks, see Section IV-D for more details. The rightmost column gives the overestimation factor, calculated as the ratio between the estimated WCET and the MOET. The estimated WCET used to calculate the overestimation factor is the one

depicted in bold face in the Table, calculated by the less pessimistic ML technique.

We observe that the estimated WCETs are never lower than MOETs. We also observe that no ML algorithm consistently outperforms the others on all benchmarks. The lowest estimated WCETs are most of the times computed by RF (8 times out of 13) and GB (3 times). The ML algorithm that computes the largest WCET estimates the most often is *Ridge* (9 times). *Pertrinet* is far the overestimated benchmark, for the rest the overestimation factor varies between 2.77 and 9.38.

We observe that benchmarks with deeply nested loops suffer from the most important WCET overestimations. This comes from the way caches are accounted for in WE-HML, which is by construction pessimistic: (i) we consider that every memory access within a loop nest may pollute the cache; (ii) the referenced addresses within a loop nest are not computed (only their number), thus the same address may be counted several times; (iii) the impact of pollution is evaluated by searching for the references having the largest impact. In comparison, the overestimation of WCETs for the benchmarks with a loop nesting level of 1 is moderate.

A more detailed analysis is now given for benchmark *binarysearch*. This benchmark is sufficiently simple to make sure that the pessimism of WCET estimates only comes from our technique: this example has obvious worst-case input, constant loop bounds, and no infeasible path. Figure 5 depicts the MOET and ML-predicted WCET for all selected ML algorithms.

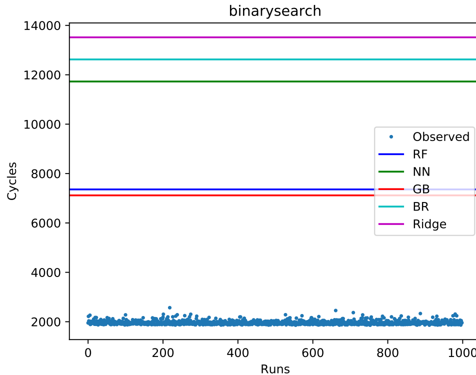

Fig. 5. ML-predicted WCETs versus observed execution times for *binarysearch*

For *binarysearch*, the smallest WCET estimate is obtained by *GB* followed tightly by *RF* and the highest is obtained by *Ridge*. The pessimism for such an application with a loop nesting level of 1 is moderate.

Although we did not observe any underestimated WCET estimates using WE-HML, one may wish to change the way the WCET of basic blocks are estimated during training, by using a lower exceedance probability. Experiments with a probability of 10^{-6} resulted in an augmentation of 35% of estimated WCETs, on average for all benchmarks.

As far as the duration of WCET estimation is concerned, we observed WCET prediction durations of around thirty seconds for most programs. This duration looks very reasonable to us for our non-optimized code for WCET estimation (call of a Python script for each basic block, parameter passing using files, de-serialization of the ML algorithm for each basic block). We observed that RF is by far the most time-consuming algorithm. The other algorithms are comparatively much faster.

B. Benefits of cache modeling

One of the benefits of WE-HML is to account for caches, by predicting two different WCETs per basic block: one for its first execution (*cold cache*) and one for the next executions (*warm cache*). Table IV gives for all benchmarks the improvement of WCET estimates obtained by accounting for caches in WE-HML, compared to the WCET estimation technique that systematically considers a cold cache (hereafter $W_nocache$). The improvement, expressed as a percentage, is calculated by formula $\frac{W_nocache - WEHML}{W_nocache}$.

As expected the numbers show a significant improvement brought by cache management (65% on average for all ML algorithms and all benchmarks). Benchmark *Pertrinet* does not benefit at all from cache management: its code contains a loop, but the amount of data accessed in the loop is so big that even when considering caches all the cache is considered as polluted by our analysis (and is actually polluted at run-time).

C. Comparison with a hybrid WCET estimation technique

In this section, we compare WE-HML with a cache-agnostic hybrid technique that uses the highest measurement for each basic block for WCET estimation. We did not compare with static WCET estimation since there is no publicly available description of the processor we are targeting, and most importantly because we are specifically targeting processors reaching the limits of static WCET estimation.

Comparing WE-HML with measurement-based hybrid approaches [7], [31] on a large set of benchmarks is difficult, because such techniques have to automatically introduce instrumentation code in the benchmark under study to measure the execution time of small code snippets. Since introducing instrumentation code is a time-consuming task, as a preliminary experiment, we performed a comparison on one program only, using manual instrumentation. The program, given in Listing 4, implements edge detection in an image. To limit the cost of insertion of instrumentation, only the main (and longest to execute) basic block was instrumented.

Listing 4. Edge detection program

```
void edge (char in[T][T],char out[T-1][T-1]) {
  for (i=0;i<T-1;i++) {
    for (j=0;j<T-1;j++) {
       a1=in[i][j]-in[i+1][j+1];
       a1=(a1+(a1>>31))^(a1>>31);
       a2=in[i][j+1]-in[i+1][j];
       a2=(a2+(a2>>31))^(a2>>31);
       out[i][j] = a1+a2;
  }}}
```

The hybrid technique used as a baseline measures the execution time of basic blocks and then applies IPET with

TABLE III
ESTIMATED WCET OBTAINED BY WE-HML VERSUS MOET.

Benchmark	RF	NN	GB	BR	Ridge	MOET	Overestimation factor
binarysearch	7358	11728	**7117**	12622	13517	2568	2.77
bsort	**3362849**	5251155	4463131	9555110	10225058	358380	9.38
countnegative	102506	99415	108291	**79818**	87545	29720	2.69
crc	**277623**	329852	298225	289192	302788	66867	4.15
expint	**27704**	35933	28353	60420	61358	6122	4.52
fdct	**26193**	40328	29084	34523	37461	8877	2.95
fir	40565	50510	**37570**	82433	87648	7646	4.91
h264_dec	**2941623**	3649120	3405644	4126177	4506618	426327	6.9
insertsort	12293	15322	**12095**	16858	18584	3042	3.98
jfdctint	**31969**	40103	35706	38910	41611	8070	3.96
matrix1	**65679**	102079	65911	95697	106144	21380	3.07
ns	190940	**183002**	185426	367042	370772	22018	8.31
petrinet	**77039**	175362	92620	157400	167268	3329	23.15

TABLE IV
IMPROVEMENT (DECREASE) OF ESTIMATED WCET RESULTING FROM CACHE MANAGEMENT

	RF	NN	GB	BR	Ridge	Avg
binarysearch	74%	70%	78%	61%	62%	69%
bsort	69%	69%	64%	27%	32%	52%
countnegative	85%	87%	86%	0%	87%	62%
crc	88%	91%	89%	90%	90%	90%
expint	80%	82%	82%	65%	68%	75%
fdct	76%	79%	77%	77%	78%	78%
fir	51%	69%	65%	37%	41%	53%
h264_dec	72%	81%	73%	73%	74%	75%
insertsort	81%	89%	86%	84%	85%	85%
jfdctint	77%	80%	78%	73%	75%	77%
matrix1	84%	84%	86%	81%	81%	84%
ns	58%	68%	63%	21%	28%	48%
petrinet	0%	0%	0%	0%	0%	0%

the largest observed value, with no attempt to account for the different execution contexts of basic blocks. This technique corresponds to the technique described by Kirner et al. in [7] with instrumentation at the basic block level.

TABLE V
COMPARISON WITH HYBRID METHOD

	BB first (cycles)	BB next (cycles)	WCET (cycles)	$\frac{WCET}{MB-Hybrid}$
RF	2160	227	76605568	3.10
NN	3247	269	91026704	2.61
GB	2595	263	87089720	2.73
BR	2381	283	114662216	2.07
Ridge	2660	286	112299136	2.11
MB-Hybrid	451	NA	237379792	1.0

Experimental results (see Table V) give for the studied basic block the WCET estimated for the different ML algorithms (two values for first and next iterations). The third line gives the WCET estimated using IPET. The last line in the table contains the ratio $\frac{Hybrid}{WE\text{-}HML}$. We observe from Table V that cache-agnostic hybrid methods are as expected more pessimistic than WE-HML, with WCET estimates 2.5 times higher than WE-HML on average. Furthermore, as already mentioned before, WE-HML does not need code instrumentation and does not suffer from the code coverage issue.

TABLE VI
R2 SCORES OF SCIKIT ML ALGORITHMS ON BASIC BLOCKS, DEPENDING ON TECHNIQUE USED FOR ESTIMATING THE WCET OF BASIC BLOCKS AND POLLUTION VALUE

	MOET			pWCET 10^-3		
Algorithm	1	16	512	1	16	512
RF	0.070	0.484	0.828	0.728	0.431	0.883
NN	0.073	0.461	0.831	0.725	0.410	0.877
GB	0.080	0.482	0.830	0.735	0.426	0.884
BR	0.077	0.467	0.827	0.722	0.415	0.874
Ridge	0.076	0.467	0.827	0.722	0.415	0.874

D. Prediction of WCETs of basic blocks

WCET prediction at program level obviously depends on ability of the ML algorithms to predict WCETs at the basic block level. This ability to predict the WCET of basic blocks is evaluated in Table VI by analyzing the *r2_score* of the ML algorithm (or the coefficient of determination) as provided by Scikit [18]. The higher the score, the better the prediction, with a best possible value of 1. The scores are given for the two different ways of calculating the WCETs of basic blocks from measurements, that are later used for training (MOET and pWCETs with exceedance probability of 10^{-3}, see Section III-C), and then per pollution values (1, 16, 512).

We observe that training the ML algorithms using the MOET of basic blocks may lead to very low scores. An analysis of the training data made us attribute this phenomenon to rare but very high timing outliers in the measurements, probably coming from the operating system activity. Probabilistic techniques such as pWCET 10^-3, by construction, are more robust to the presence of such outliers, that they eliminate if rare enough. We did not notice any significant difference in the learning scores obtained by the different algorithms in the same scenario. pWCET 10^-6 was observed to have slightly higher R2 scores than pWCET 10^-3, but significantly higher WCETs (35% on average on the benchmarks).

The best scores are observed for the configurations with a

low pollution value, and for those with the highest pollution value, which is expected since the execution times in these situations have low variability. With intermediate pollution values, the scores are lower. The worst scores are obtained with pollution values 2, 4 and 8, and then the scores improve when the pollution value increases. With small pollution values $(2, 4, 8, 16)$, the high variability of timings, comes from the fact that it is harder to exercise the worst-case cache collisions by randomly writing to memory.

V. Related work

The technique we propose in this paper is a *hybrid* technique, in which the longest path is estimated using static techniques, and the timing of basic blocks is estimated using an empirical technique. In contrast to most existing hybrid techniques such as [7]–[10], [31], [32], WE-HML does not use measurements to estimate the WCET of basic blocks, but instead uses a timing model learnt using ML techniques. As a consequence, coverage of basic blocks and timing instrumentation is not an issue in WE-HML. Among hybrid techniques, to our best knowledge only the techniques by Stattelmann and Martin [9] and by Dreyer et al. [10] support the variability of execution duration induced by caches. However, these two measurement-based hybrid techniques demand reverse engineering of timing traces to separate the different execution contexts of basic blocks.

Regarding the definition of timing models of hardware, Asavoae et al. [33] addresses the issues of formal co-validation of hardware and software timing models of safety-critical systems. The approach defines a formal specification of the timing behavior of hardware for a very simple processor. WE-HML is targeted for more complex processors, for which extracting the timing behavior of the processor is out of reach. For static WCET estimation techniques, static analysis techniques have been designed to obtain safe timing estimations of the WCET of basic blocks in the presence of cache hierarchies [2], [34], [35]. Compared to these techniques, WE-HML requires little knowledge of the memory hierarchy (size of last level cache only).

Mendis et al. present in [36] a technique to estimate the average-case steady-state (i.e. best-case) performance of the software for x86 architectures, using hierarchical neural networks. In contrast to their work, we focus on the prediction worst-case performance instead of average-case.

In the rest of this section, we compare our work to the existing literature using machine learning techniques for WCET estimation [15]–[17].

The research presented in [17] estimates WCETs of portions of code that are larger than basic blocks. This means that the machine learning algorithms, in addition, to learning the timing of hardware, have also to learn the longest execution path in programs. In contrast, we use ML for timing estimation of basic blocks only and use the safe standard WCET calculation technique IPET for identifying the longest path.

No research amongst [15]–[17] asked explicitly the ML algorithms to learn the behavior of architectural elements such as caches. We propose a first attempt to take into account the cache hierarchy. Compared to [15]–[17] we perform our analysis at machine code level while the referenced research operates on intermediate or source code. As a result, the quality of predictions is better, as the code analyzed will faithfully reflect what is actually executed by the hardware.

Lisper and Santos propose in [32] a hybrid approach where, for a given program, the WCET of basic blocks are estimated from a set of end-to-end-measurements, using a technique close to linear regression. Contrary to our work, the method does not model cache effects, and the learning process is based on end-to-end measurements on the program under analysis itself, rather than learning the timing from code snippets like basic blocks.

Courtaud et al. [37] propose the use of ML to estimate interference delays in shared memory multi-core architectures. In contrast to their work, we focus on single-core processors. Their technique could complement ours for validating timing on multi-cores. Similarly, the research by Griffin et al. [38], could complement our work to support multi-cores.

Finally, [15], [17] train their ML algorithms on benchmarks of the literature, which contain a limited set of programs. In contrast, WE-HML and [16] use a C code generator to automatically generate training data. Thus, the variability and the amount of the data trained on is controlled. Our basic block generator is inspired by Gene [21] that generates LLVM IR-code for WCET estimation. The main difference with Gene and [16] is that the code generator produces basic blocks and not entire programs.

VI. Conclusion

This paper has proposed WE-HML, a hybrid WCET estimation technique, for which the WCET of basic blocks is estimated using ML techniques in order to avoid the need for precise knowledge of the architecture. Compared to existing literature using ML for WCET estimation, WE-HML accounts for data caches, and predictions operate at the machine code level. Experimental results have shown that amongst the ML algorithms we have experimented with, none of them consistently outperforms the others on all benchmarks. Although WE-HML does not offer safety guarantees, we observe that predicted WCETs are always higher than any observed execution times for all benchmarks. Furthermore, cache modeling allows an average improvement of WCET estimates of 65% compared to a cache-agnostic equivalent of WE-HML. Finally, WCET estimates for all benchmarks are calculated in seconds for most benchmarks.

Although giving encouraging results, the proposed method could be improved in several directions. The main direction for future work is to improve our analysis of the impact of cache pollution, for example by considering instruction caches or improving the management of loop nests. More generally, the concept of *execution context* for the basic block has to be extended beyond the simple *pollution value* considered in this paper, to consider code properties having an impact on timing.

ACKNOWLEDGMENTS

The authors thank Tom Bachard, Victor Careil, Antoine Gonon and Grégoire Pacreau for their initial work, performed in the context of a student project. The authors are also very grateful to Claire Pagetti, Julia Lawall and Élisa Fromont for their fruitful comments on earlier drafts of the paper.

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[2] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.

[3] X. Li, A. Roychoudhury, and T. Mitra, "Modeling out-of-order processors for WCET analysis," *Real-Time Systems*, vol. 34, no. 3, pp. 195–227, 2006.

[4] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Real-Time Systems*, vol. 18, no. 2/3, pp. 249–274, 2000.

[5] F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernández, J. Abella, and T. Vardanega, "Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey," *ACM Comput. Surv.*, vol. 52, no. 1, pp. 14:1–14:35, 2019.

[6] J. Deverge and I. Puaut, "Safe measurement-based WCET estimation," in *International Workshop on WCET Analysis*, 2005.

[7] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, "Using measurements as a complement to static worst-case execution time analysis," *Intelligent Systems at the Service of Mankind*, vol. 2, 01 2006.

[8] A. Betts and G. Bernat, "Tree-based WCET analysis on instrumentation point graphs," in *Ninth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2006, pp. 558–565.

[9] S. Stattelmann and F. Martin, "On the use of context information for precise measurement-based execution time estimation," in *International Workshop on WCET Analysis*, 2010, pp. 64–76.

[10] B. Dreyer, C. Hochberger, A. Lange, S. Wegener, and A. Weiss, "Continuous non-intrusive hybrid WCET estimation using waypoint graphs," in *International Workshop on WCET Analysis*, 2016, pp. 4:1–4:11.

[11] B. Lesage, S. Law, and I. Bate, "TACO: an industrial case study of test automation for coverage," in *26th International Conference on Real-Time Networks and Systems, RTNS*, 2018, pp. 114–124.

[12] J.-P. Blanquart, J.-M. Astruc, P. Baufreton, J.-L. Boulanger, H. Delseny, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, J. Machrouh, P. Quéré, and B. Ricque, "Criticality categories across safety standards in different domains," in *Embedded Real Time Software and Systems (ERTS)*, 2012.

[13] , "Raspberry pi platforms," 2020. [Online]. Available: https://www.raspberrypi.org/

[14] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wägemann, and S. Wegener, "Taclebench: A benchmark collection to support worst-case execution time research," in *International Workshop on WCET Analysis*, 2016, pp. 2:1–2:10.

[15] T. Huybrechts, S. Mercelis, and P. Hellinckx, "A new hybrid approach on WCET analysis for real-time systems using machine learning," in *International Workshop on WCET Analysis*, 2018, pp. 5:1–5:12.

[16] P. Altenbernd, J. Gustafsson, B. Lisper, and F. Stappert, "Early execution time-estimation through automatically generated timing models," *Real-Time Systems*, vol. 52, no. 6, pp. 731–760, 2016.

[17] A. Bonenfant, D. Claraz, M. D. Michiel, and P. Sotin, "Early WCET prediction using machine learning," in *International Workshop on WCET Analysis*, 2017, pp. 5:1–5:9.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *DAC: 32nd ACM/IEEE conference on Design automation*, 1995, pp. 456–461.

[20] D. Hardy, B. Rouxel, and I. Puaut, "The Heptane static worst-case execution time estimation tool," in *International Workshop on WCET Analysis*, 2017, pp. 8:1–8:12.

[21] P. Wägemann, T. Distler, T. Hönig, V. Sieh, and W. Schröder-Preikschat, "Gene: A benchmark generator for WCET analysis," in *International Workshop on WCET Analysis*, 2015, pp. 33–43.

[22] I. Bate, D. Griffin, and B. Lesage, "Establishing confidence and understanding uncertainty in real-time systems," in *ACM International Conference on Real-Time Networks and Systems*, 2020, p. 67–77.

[23] H. Li, I. Puaut, and E. Rohou, "Tracing flow information for tighter WCET estimation: Application to vectorization," in *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2015, pp. 217–226.

[24] M. Dardaillon, S. Skalistis, I. Puaut, and S. Derrien, "Reconciling compiler optimizations and wcet estimation using iterative compilation," in *IEEE Real-Time Systems Symposium (RTSS)*, 2019.

[25] H. Falk and P. Lokuciejewski, "A compiler framework for the reduction of worst-case execution times," *Real-Time Systems*, vol. 46, no. 2, pp. 251–300, 2010.

[26] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution," in *27th IEEE Real-Time Systems Symposium (RTSS)*, 2006, pp. 57–66.

[27] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. D. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim, "PROARTIS: probabilistically analyzable real-time systems," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 2s, pp. 94:1–94:26, 2013.

[28] L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart, "On the sustainability of the extreme value theory for wcet estimation," in *International Workshop on WCET Analysis*, 2014.

[29] F. Reghenzani, G. Massari, W. Fornaciari, and A. Galimberti, "Probabilistic-wcet reliability: On the experimental validation of evt hypotheses," in *International Conference on Omni-Layer Intelligent Systems*, 2019, pp. 229–234.

[30] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow: Concepts, Tools, and Techniques to Build Intelligent Systems*. OŔeilly, 2019.

[31] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand, "TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis," in *International Workshop on WCET Analysis*, ser. OpenAccess Series in Informatics (OASIcs), vol. 72, 2019, pp. 1:1–1:11.

[32] B. Lisper and M. Santos, "Model identification for wcet analysis," in *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2009, pp. 55–64.

[33] M. Asavoae, I. Haur, M. Jan, B. B. Hedia, and M. Schoeberl, "Towards formal co-validation of hardware and software timing models of cpss," in *Cyber Physical Systems. Model-Based Design - 9th International Workshop, CyPhy'19, and 15th International Workshop, WESE Revised Selected Papers*, 2019, pp. 203–227.

[34] D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," in *IEEE Real-Time Systems Symposium (RTSS)*, 2008, pp. 456–466.

[35] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *LITES*, vol. 3, no. 1, pp. 05:1–05:48, 2016.

[36] C. Mendis, A. Renda, S. P. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *36th International Conference on Machine Learning, ICML*, 2019, pp. 4505–4515.

[37] C. Courtaud, J. Sopena, G. Muller, and D. G. Pérez, "Improving prediction accuracy of memory interferences for multicore platforms," in *IEEE Real-Time Systems Symposium, RTSS*, 2019, pp. 246–259.

[38] D. Griffin, B. Lesage, I. Bate, F. Soboczenski, and R. I. Davis, "Forecast-based interference: modelling multicore interference from observable factors," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, 2017, pp. 198–207.