CAWET: Context-Aware Worst-Case Execution Time Estimation Using Transformers

Abderaouf N Amalou, Elisa Fromont, Isabelle Puaut

CAWET: Context-Aware Worst-Case Execution Time Estimation Using Transformers

Abderaouf N Amalou
Univ. Rennes, INRIA, CNRS, IRISA, France

Elisa Fromont
Univ. Rennes, IUF, INRIA, CNRS, IRISA, France

Isabelle Puaut
Univ. Rennes, INRIA, CNRS, IRISA, France

Abstract

This paper presents CAWET, a hybrid worst-case program timing estimation technique. CAWET identifies the longest execution path using static techniques, whereas the worst-case execution time (WCET) of basic blocks is predicted using an advanced language processing technique called Transformer-XL. By employing Transformers-XL in CAWET, the execution context formed by previously executed basic blocks is taken into account, allowing for consideration of the micro-architecture of the processor pipeline without explicit modeling. Through a series of experiments on the TacleBench benchmarks, using different target processors (Arm Cortex M4, M7, and A53), our method is demonstrated to never underestimate WCETs and is shown to be less pessimistic than its competitors.

2012 ACM Subject Classification Computer systems organization → Real-time system architecture

Keywords and phrases Worst-case execution time, machine learning, transformers, hybrid technique

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.7

1 Introduction

The Worst-case execution time (WCET) of a task is its maximum execution time when varying its input data and hardware state. Knowledge of the WCET of all tasks in a system allows schedulability analysis techniques to demonstrate that all tasks will meet their timing requirements in real-time systems. The challenge addressed in this paper is to estimate WCETs for Commercial Off The Shelf (COTS) processors, for which the micro-architecture details are not fully known.

WCET estimation techniques can be divided into three broad categories [36]: *static*, *measurement-based*, and *hybrid* techniques.

Static techniques (ST, e.g., [16, 3]) operate on the Control Flow Graph (CFG) of the task, extracted from its binary code. The nodes in the CFG are Basic Blocks[1] (BB), and the edges represent the control flow between the BB. Static techniques proceed in two phases: in the first phase, the WCET of each BB is estimated using abstractions of the hardware state; in the second phase, the whole program's WCET is calculated by finding the worst path inside the CFG (e.g., this is achieved by employing the commonly used implicit path enumeration technique – IPET – [36]). Although the static techniques produce safe WCET estimates, using hardware abstractions on complex micro-architectures will inevitably lead to state explosion. Moreover, each new architecture demands the design of a new hardware abstraction, which is time-consuming and error-prone (especially without the processor's micro-architectural details).

[1] A basic block is defined as a sequence of instructions with a single entry point at the beginning and a single exit point at the end, without any branching or jumping to other instructions within the block.

Measurement-based techniques (MBT) (e.g., [9]) are empirical techniques that run the program end-to-end with varied input data and hardware states to gather measurements. The WCET is then estimated from the measurements by either returning the largest observed timing (with a configurable safety margin) or using statistical techniques such as extreme value theory to infer a probabilistic WCET from the observed values [29, 30]. Unless the worst input and hardware state are found, techniques in this category may produce unsafe results.

Hybrid techniques (HT) [4, 32, 11, 20, 21] combine the benefits of ST and MBT: the longest path is safely identified using techniques from ST, like IPET; measurements are used at the BB level, avoiding the costly and error-prone design of hardware abstractions. However, using measurements at the BB level in hybrid methods raises code coverage issues: each BB has to be executed at least once, and each BB's worst-case scenario must be covered.

In recent works, machine learning (ML) techniques are used in HT instead of measurements to predict the WCET of BBs [5, 18, 17, 24, 25, 2]. These techniques, named HT-ML in the following, train an ML model (e.g., neural network) on a large dataset of BB whose WCET is known. The ML model is then used to predict the WCET of previously unseen BB. HT-ML techniques have the following benefits:

(i) The time-consuming phase of HT-ML (training) is executed only once (per architecture) and does not need any design of a hardware abstraction like in ST.

(ii) Although the training phase may be long, prediction is fast and does not require thousands of measurements per BB.

(iii) HT-ML can process large amounts of execution scenarios for BB and identify patterns, allowing more accurate predictions.

Nevertheless, the current HT-ML methods use oversimplified code characterization. The features used for learning and prediction abstract too much information from the machine code, causing information that impacts timing to be lost. For example, not considering the ordering of machine instructions in a BB will make the technique unable to accurately learn the impact of pipelines on timing.

In this paper, we propose a novel HT-ML WCET estimation technique called CAWET, for **C**ontext-**A**ware **W**orst-case execution time **E**stimation using **T**ransformers. This technique uses the advanced machine learning algorithm Transformers-XL [8]. Unlike other HT-ML methods, which only consider static features, CAWET considers the internal dependencies within each BB and the context surrounding it when estimating its WCET. This is performed by treating the sequence of instructions in a BB as a natural language, where the timing of a BB depends not only on its own sequence of instructions but also on the sequence of BBs executed before it.

CAWET consists of two main stages: *training* and *deployment* (or *estimation*). As in all systems using Transformers, the Transformer model is first pre-trained in the learning phase to comprehend the vocabulary (in our context, assembly language). Then, the model is fine-tuned using extensive measurements on various basic blocks extracted from real codes. In this fine-tuning stage, the model learns how to calculate the WCET of each basic block by considering the context surrounding the block (previously executed BBs). During the estimation stage, the WCET of each BB is determined for all bounded-length contexts leading to the BB, extracted from the program's CFG. The maximum timing estimate for these contexts is then selected as the WCET of the basic block and used by IPET to calculate the WCET of the overall program.

CAWET is easy to deploy, as the training has to be done only once. Consideration of pipeline effects is performed automatically because of the consideration of the execution context of all basic blocks.

CAWET is evaluated on processors of varied complexity, including the basic pipeline-only cortex-M4, the more advanced cortex-M7 that features a cache, and the even more sophisticated cortex-A53. The quality of WCET estimates produced by CAWET is compared to those produced by WE-HML, the HT-ML technique closest to CAWET [2], on 13 programs from the TACLeBench benchmark suite [13]. Our results show that CAWET produces better estimates than its competitors on more diverse architectures.

Our contributions are:

- A new hybrid timing ML-based WCET analysis technique that uses Transformers-XL to estimate the WCET of basic blocks and considers dependencies between instructions.
- We take into account the execution context that surrounds the BB under analysis by automatically exploring all bounded-length paths that leads to it.
- We provide an empirical study on different targets and techniques. Our results show that this complex ML method is well-suited for timing estimation, with an average error of 23.8%, 102.2%, and 62.4%, on the Cortex M4, Cortex M7, and Cortex A53 processors, respectively.

The rest of this paper is organized as follows. Section 2 presents the CAWET HT-ML technique. The experimental methodology for evaluating it is detailed in Section 3, and experimental results are given in Section 4. Section 5 compares our approach to related techniques. We conclude in Section 6.

2 CAWET: Context-Aware WCET estimation using Transformers

CAWET is a hybrid context-aware WCET estimation technique that predicts an in-context WCET of individual basic blocks and then uses the predictions to calculate the overall program's WCET. A high-level overview of CAWET is given in Section 2.1. The two main phases of CAWET: training (using Transformers-XL) and prediction (i.e., deployment), are then respectively presented in Sections 2.2 and 2.3.

2.1 Overview of CAWET

CAWET consists of two main stages: *training* and *deployment* (or *estimation*). Both stages operate on individual basic blocks (BB) and account for the execution context of the BB under study (i.e., the sequence of BBs executed before it). CAWET relies on Transformers-XL, originally used in natural language processing, for their ability to learn long-term dependencies between words. In CAWET, the language under study is a sequence of BBs, each composed of a sequence of assembly instructions. The overall structure of CAWET is depicted in Figure 1.

In the training phase (left block of Figure 1), the Transformer model is first pre-trained on real programs to learn the vocabulary of the language it will process (in our context, assembly language) as it is usually done for large language models [10]. Then, the model is fine-tuned using extensive measurements on a large set of BBs extracted from real code. In this fine-tuning stage, the model learns how to calculate the WCET of each BB by considering the context surrounding it (i.e., previously executed BBs).

During the estimation stage (right block of Figure 1), the WCET of each BB is determined. Since there might be different execution paths leading to the BB under study, prediction operates on the set of contexts corresponding to these paths, with care taken to avoid combinatorial explosion, as further explained in Section 2.3. The prediction phase first computes the list of contexts of the BB under study (BB number 8 in the Figure). The result

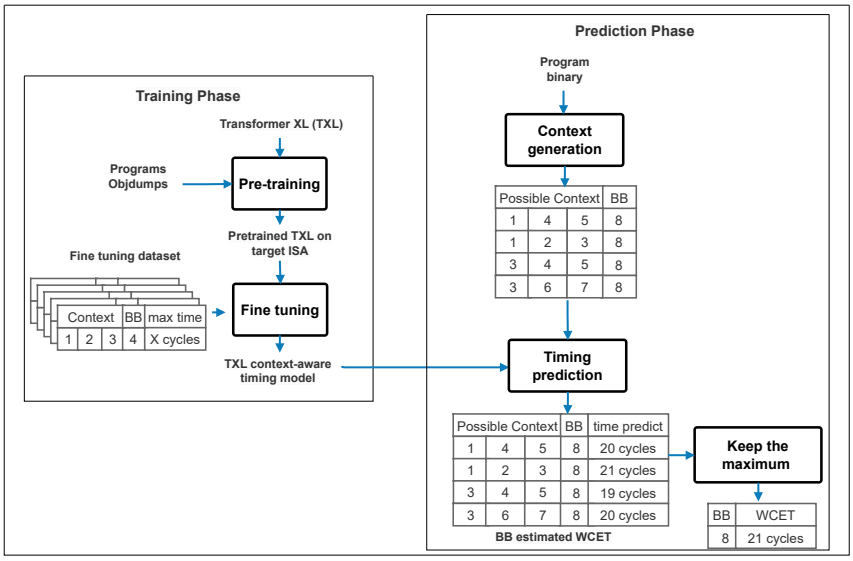

Figure 1 Overview of CAWET.

in the example is a list of 4 contexts, made of the sequence of BBs executed before BB 8: $(1, 4, 5), (1, 2, 3), (3, 4, 5)$, and $(3, 6, 7)$. The timing of BB 8 is estimated for each context. The maximum timing estimate is then selected as the WCET of the BB and used by IPET to calculate the WCET of the overall program.

2.2 Training phase using Transformers-XL

A *transformer* is a neural network architecture originally designed for natural language processing, which can perform tasks such as language translation, text summarizing, and text-to-speech. It was first proposed in [35], and one of its main advantages is using self-attention mechanisms that enable the model to weigh different parts of the input data when making predictions. However, as defined in [35], the original transformer architectures have a fixed-length context window and may struggle to handle sequential data with long-term dependencies. To address this limitation, *Transformers-XL* (TXL) [8] were introduced. A TXL is a variation of the transformer architecture that uses a so-called *memory-augmented attention* to better remember and utilize information from earlier in the sequence. We use a TXL architecture in CAWET because it improves the ability of the transformer to handle long-term dependencies, which is necessary for handling long sequences of code.

Estimating the WCET of a given BB given its context is performed by first processing the context (formed by the BB executed before the analyzed BB as well as the analyzed BB), followed by processing the BB under analysis. This results in two embedding matrix representations (a global attention matrix for the context and a local attention matrix for the BB under analysis) that are then concatenated. The resulting embedding representation is given as input to a fully connected layer, producing a single scalar value (the timing estimate for the analyzed BB). Figure 4 provided in the Appendix illustrates this process.

The training of a TXL consists of two stages (*pre-training* and *fine-tuning*). During the pre-training stage, the TXL is trained to learn the structure of assembly instructions in text format using self-supervised learning. This classical self-supervised learning phase [10] is achieved by masking random operations or operands in the sequence and (pre)training the model to reconstitute (i.e., predict) them as output. To perform this pre-training phase,

thousands of disassembled binary programs are used without needing labeled information. Details about the hyper-parameters of the TXL architecture are provided in Table 10 in the Appendix.

In the fine-tuning stage, a set of programs, the target processor, and a measurement tool are required. BBs execution time is measured using the measurement tool. Then, the instruction sequences are tokenized using *sentence piece* [23], a well-known tokenization technique trained in our work on the target assembly instructions. The training dataset for the fine-tuning stage is then built using the maximum observed timing of each BB, the tokenized BB, and its context. Contexts have a maximum size; the context size, expressed as a number of basic blocks, is a hyperparameter of the Tranformer-XL.

2.3 Prediction phase

CAWET predicts the WCET of BBs by considering their execution context. The results from CAWET can then be used by a static WCET estimation tool. Section 2.3.1 presents the concepts and notations CAWET relies on. Section 2.3.2 then details the context generation. Section 2.3.3 describes how the WCET of a BB is obtained from the predictions and the overall WCET of the program is finally calculated.

2.3.1 Concepts and notations

The concepts and notations used in CAWET are standard concepts used in compilers. They are illustrated in Figure 3, which will be reused later to illustrate how CAWET works.

▶ **Definition 1** (Control Flow Graph). *A Control flow graph (CFG) is a directed graph where each node represents a BB, and each edge represents the control flow from one BB to another.*

▶ **Definition 2** (SESE regions, SESE trees). *A Single Entry Single Exit (SESE) region, as defined in [19], is a sub-graph of a CFG that can only be entered by one edge and exited by one edge. A property of SESE regions is that they can be arranged into a tree, constructed in linear time.*

An example of CFG (with 7 BBs numbered from 1 to 7), and its SESE regions is depicted in Figure 2 (A). The dotted arrow in the figure represents the back edge of the loop composed of BB 5 and 6. The SESE tree that corresponds to the CFG is depicted in Figure 2 (B). The rationale behind using SESE regions is to have subsets of the CFG that are simple enough to explore all paths exhaustively, with the overall objective of avoiding combinatorial explosion when generating the possible contexts of a BB.

▶ **Definition 3** (Cyclomatic complexity). *Cyclomatic complexity is a software metric that measures the number of independent paths through a program or a CFG [12]. It can be thought of as the number of unique paths that can be taken through the code. It is calculated using the following formula:* $Cyclomatic\_complexity(CFG) = edges - nodes + 2$

The cyclomatic complexity will be used during the prediction phase to decide which paths leading to a BB are worth exploring. The cyclomatic complexity of the SESE regions in our example is displayed in Figure 2 (B).

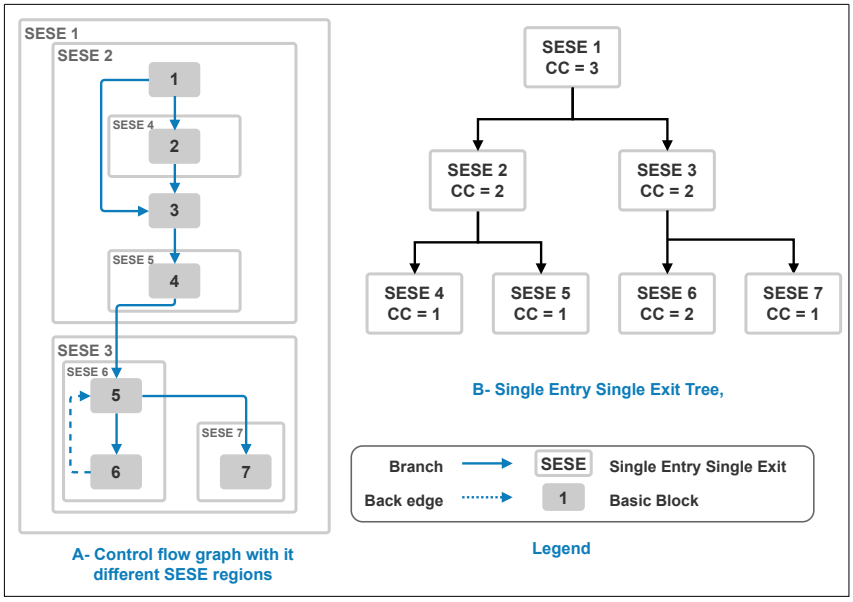

Figure 2 A CFG example transformed into a SESE tree and annotated with cyclomatic complexity.

2.3.2 Context generation

The task of finding all the possible paths in a graph may be computationally expensive. To address this issue, we use a divide-and-conquer strategy based on the SESE tree of the program. In the example of Figure 2, the root SESE region (SESE 1) represents the entire CFG. Each tree level represents a sub-SESE region (e.g., SESE 2 and SESE 3 are the children of SESE 1), with smaller and thus simpler sub-graphs.

To limit the complexity, CAWET performs an exhaustive path exploration only for the SESE regions that are simple enough (based on their Cyclomatic Complexity, CC) to allow a full path exploration. SESE selection is performed using a top-bottom traversal of the SESE tree, and the SESE regions with a value of CC strictly higher than a threshold are filtered out. Path exploration for the selected regions uses Depth-First Search [34] (DFS) to enumerate all possible paths[2]. We ensure, by construction, that the chosen SESE covers the entire input code. i.e., in situations where a SESE node cannot be analyzed due to its high CC value, we analyze all its children. Additionally, basic blocks that do not belong to any region in the tree are included to ensure complete code coverage.

This process is illustrated in Figure 3 step 1 using the CFG and SESE in Figure 2 as an example, with a CC threshold of 2. In this example, the SESE regions 2 and 3 are selected, and their paths are fully explored (step 2 in Figure 3).

Management of loops

As explained above, the enumeration of paths in SESE regions ignores the back edges of loops. Therefore, all paths in a given loop are explored only for one iteration. Obtaining the execution context of any BB to be executed after a loop requires considering several loop

[2] DFS traversal ignores loop back-edges. Loop management is described later in this Section.

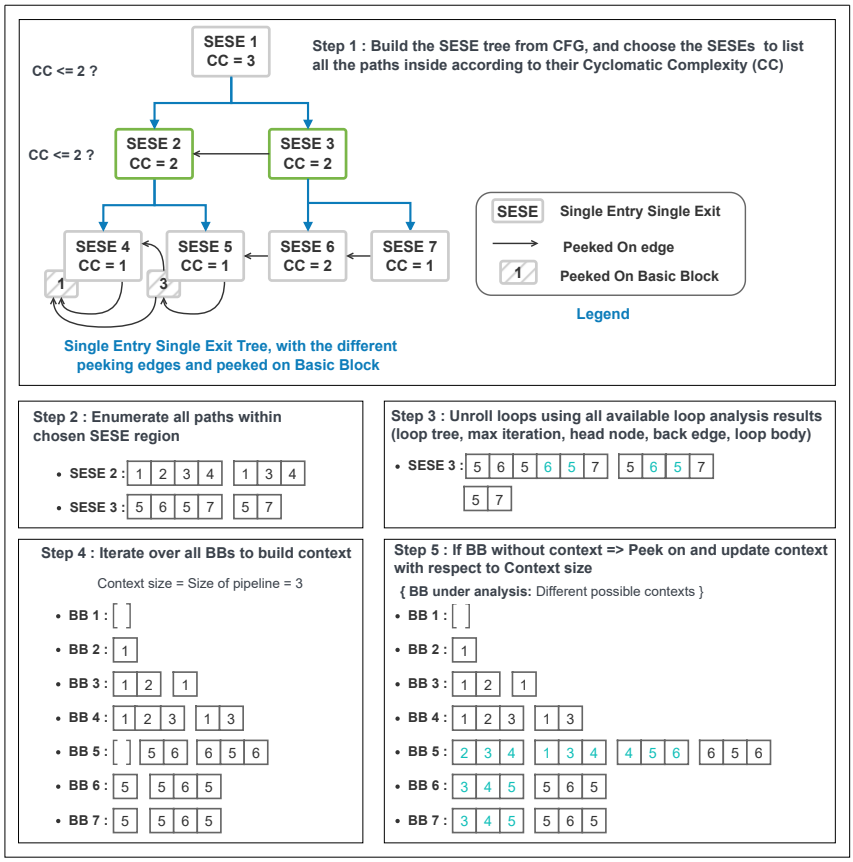

Figure 3 Example of the different steps for context generation, where the cyclomatic complexity limit is set to 2 and the context size is set to 3 BBs.

iterations. This is achieved in CAWET using (virtual) *unrolling*: the context of a loop is composed of several iterations of the loop body (from zero to the loop's maximum number of iterations).

As the path followed may differ across iterations, generating all possible contexts may lead to a combinatorial explosion. This issue is addressed by restricting the number of BBs added by the unrolling process for the loop body to a fixed value, the hyperparameter *context size* of CAWET. In the presence of nested loops, the context of the inner loops is generated first, to be further used to generate the context for outer loops. This is performed using a bottom-up traversal of the *loop nesting tree* of every CFG[3].

The result of the loop unrolling process on our example is given in Figure 3 step 3, for SESE 3. Three contexts are generated, corresponding respectively to 1, 2, and 3 executions of the loop. Note that, at this step, the size of the contexts of SESE regions may be longer than the *context size* hyperparameter.

Per BB context generation

The execution traces for the different SESE regions, after loop unrolling, are used to generate the *context list* of every basic block, as depicted in Figure 3 step 4. The size of each context is limited to the *context size* hyperparameter of CAWET.

[3] A *loop nesting tree* is a tree data structure used to represent nested loops. Each node in the tree represents a loop, and the edges between the nodes represent the nesting relationship between the loops.

In some cases, the initial nodes of some SESE sub-regions are smaller than the *context size* hyperparameter. To address this issue, we look for the preceding SESE region or BB to access the end of its traces. The peeked-on edges are shown in Figure 3; they can easily be found by looking at the end of the traces of all the BB that occur before this trace. The obtained information can then be used as context for the start nodes of the current SESE region, provided we can find a region before the current one.

As an example, Figure 3 step 5 shows that the context of BB 5 can be augmented by peeking at the execution trace of SESE 2.

2.3.3 Basic Block WCET estimation and program WCET calculation

After generating all possible limited-size contexts for each BB, we move on to estimating its WCET. This involves predicting the execution time of the BB under study for all its contexts. In an architecture without a cache, the maximum estimated time is selected as the worst-case scenario. If the target architecture includes a cache, we keep track of the two highest estimated execution values to account for cache effects. The largest value represents the first execution of the basic block within a loop, which is typically long, while the other value represents subsequent executions of the same basic block, which may be shorter[4]. The WCET of BBs is then fed into a static WCET estimation tool to calculate the WCET of the overall program using standard techniques such as IPET [36].

3 Experimental setup

This Section provides a comprehensive description of the experimental setup used to evaluate CAWET on multiple ARM Cortex targets, specifically M4, M7, and A53. The programs used to train CAWET and evaluate the quality of predictions are first described in Section 3.1. The context-agnostic baselines CAWET is compared to are presented in Section 3.2, followed by an introduction to the software and hardware environments in Section 3.3. The setups for the learning and prediction phases of CAWET are presented respectively in Section 3.4 and 3.5.

3.1 Dataset and benchmarks

CAWET training consists of two steps: (self-supervised) pre-training and fine-tuning. We have pre-trained CAWET on a large number of BBs in order for the Transformer to learn the assembly language under study, using CodeNet [28]. CodeNet is a collection of solutions submitted by the public to competitive programming websites. It contains approximately 900,000 C programs, which we cross-compile to the target architecture and disassemble using GNU binary utilities using *objdump*. The textual format produced by *objdump*, after some basic parsing (e.g., extraction of addresses, separation of BBs) allows the creation of a large pre-training set. This pre-training set is used to build a vocabulary model with *sentence piece* [23]. Once the model (sentence piece model) has been trained, it is then used to tokenize any binary programs written with the target instruction set. To fine-tune CAWET on basic blocks with their context, we have used a diverse and publicly available set of programs:

[4] Since the context size is limited, the predicted timing values may be too optimistic. We, therefore, analyze in Section 4.2 and 4.4 a technique that applies static cache analysis, and we add the overhead obtained by this analysis to the timing values produced by CAWET.

The Algorithms[5], *MiBench* [15] and *Polybench* [37]. Table 1 gives a short description of each benchmark suite, the number of programs it contains, and the total number of BBs encountered when executing the programs.

Table 1 The benchmarks used for training CAWET.

Dataset name	Description	Nb. of programs	Nb. of BB
The Algorithms	Collection of open-source implementations of a variety of algorithms implemented in C	200	12123
PolyBench	A collection of benchmarks containing static control parts. The purpose is to uniformize the execution and monitoring of kernels	30	11224
MiBench	A free, commercially representative embedded benchmark suite	14	8324
Total		244	31671

Table 2 Selected TacleBench codes used to evaluate the quality of the predictions.

Name	Description
bs	Binary search in an array
bsort	Bubble sort algorithm
countnegative	Basic counting on arrays
crc	Cyclic redundancy codes
expint	Exponential integral function
fdct	Fast discrete cosine transform.
fir	Finite impulse response filter
h264 dec	H.264 block decoding functions
insertsort	Insertion sort
jfdctint	Discrete-cosine transformation
matrix1	Generic matrix multiplication
ns	Search in 4-dimension array
petrinet	Petri net simulation

To validate the quality of the WCET predictions provided by CAWET, we use a subset of the codes from the TacleBench benchmark suite [13] whose characteristics are given in Table 2. We chose these codes because: (i) the programs are analyzable by static WCET estimation tools, and in particular, they contain loop-bound annotations; (ii) they come with input data known to trigger the worst-case execution paths; (iii) they are used in our closest competitor WE-HML [2], allowing us to compare CAWET with this work. Note that the selected TacleBench programs were not used during any of the two steps of the training phase.

3.2 Context-agnostic baselines

CAWET is evaluated by comparing it to two context-agnostic WCET predictors. The first one is a Multi-Layer Perceptron regressor (loosely called a neural network (NN)). Although not a naive approach, the neural network is a feed-forward architecture that does not incorporate sequential information and requires a fixed-size input. Our implementation of the NN employs a total of 233 static features of the basic blocks as input, including the proportion of different machine instruction types (e.g., MOV, ADD, LDR). We used a greedy search algorithm to determine optimal hyperparameters for the NN, including the number of

[5] Available here: `https://github.com/TheAlgorithms/C`

hidden layers, optimizer, learning rate, and loss function. Based on the validation dataset, the ideal parameters were determined to be hidden layer sizes=(512, 256, 128), learning rate='adaptive', learning rate init=0.001, solver='adam'. The other baseline CAWET is compared with is WE-HML, a hybrid ML-based WCET estimation technique presented in [2]. The best performing ML algorithm of [2] (Neural Network trained to account for cache effects) is used. CAWET is compared to WE-HML for the Cortex A53 processor only, a processor for which the results of WE-HML were available.

3.3 Hardware and software setups

Accurate timing values must be employed whenever possible when training and validating CAWET, and the method used to obtain the timing values should not interfere with the execution of the code, a phenomenon commonly known as the *probe effect*. CAWET either uses a hardware-based approach or a software solution when the hardware-based solution is not accessible. The hardware solution leverages the Joint Test Action Group (JTAG) interface. The J-Trace Pro trace solution from Segger [31] is used to connect to the JTAG interface of the target processor (in our case Cortex-M4 and Cortex-M7), in conjunction with Ozone [14], a cross-platform debugger and performance analyzer. Ozone generates execution traces that provide the value of the cycle counter, the instruction's address, opcode, and operands, as well as the corresponding assembly code for each instruction. The software solution involves adding code instrumentation to measure the execution time of individual basic blocks (BB) in a program. To provide context and assembly code for the timed BB, we retrieve the execution trace using GDB (the GNU Debugger). The software solution is only used when no JTAG interface is available since it is prone to probe effects and requires significant human effort to implement.

Our experiments are performed on various Arm processors, whose characteristics are summarized in Table 3. We initially focus on the Cortex-M4 processor, which has a simple in-order pipeline with three stages and no cache. This processor allows us to validate our method on a deterministic processor with precise timing measurements through the JTAG interface. Then, we evaluate our approach to the more advanced Cortex-M7 processor. This processor features a 6-stage in-order pipeline, data and instruction caches, and a branch predictor. Finally, we use a more complex processor, Cortex-A53, which is hosted in a Raspberry Pi 3. This superscalar processor has two data and instruction cache levels: an 8-stage in-order pipeline and a branch predictor. The Cortex-A53 has no JTAG interface; the reading of the cycle counter is used for the timing measurements. Using this commercial off-the-shelf (COTS) hardware is part of the experiments in the WE-HML approach [2].

Table 3 Summary of the processors used and their micro-architectural features.

Target	Measurement solution	OS?	Pipeline/#stages	Branch predictor	Cache memory and proprieties
Cortex-M4	Hardware (JTAG)	Baremetal	In-order/3	No	No
Cortex-M7	Hardware (JTAG)	Baremetal	In-order/6	Yes	Yes data and instruction cache, L1, random replacement policy
Cortex-A53 (also used in [2])	Software	Linux	In-order/8	Yes	Yes data and instruction cache, L2, random replacement policy

3.4 Setup for the learning phase

PyTorch was used to implement the learning models, which were then trained on a Tesla V100. Each setting (processor) required two days for CAWET training: 1,5 days for pre-training and 0,5 days to fine-tune the model. To avoid underestimating execution times, we employed the

Root Mean Squared Logarithmic Error (RMSLE) loss function provided in Equation 1, which tends to penalize underestimations more heavily than overestimations. We also incorporated an additional penalty for predictions that underestimated the execution time, according to Equation 2. We artificially modify the target value in the loss when the prediction is too low. When computing the loss, this is done by increasing the target with the predicting error $(target - prediction)$.

$$RMSLE(target, predict) = \sqrt{(\log(target + 1) - \log(predict + 1))^2} \tag{1}$$

$$UsedTarget = \begin{cases} target & if \quad target \leq prediction \\ target + (target - prediction) & if \quad target > prediction \end{cases} \tag{2}$$

3.5 Setup for the prediction phase

The CFG, the SESE tree, and the loop tree is generated by the Heptane WCET estimation tool [16]. These structures are used to construct the list of contexts for each BB. Then, we predict the WCET for each BB using CAWET. Finally, we employ Heptane's IPET to determine the overall WCET of the program.

To create the contexts, we opted for a cyclomatic complexity of 5, as this value has been shown empirically to generate paths within a reasonable amount of time (less than five minutes to generate traces for each basic block in the 13 programs previously described). Since the best context size varies across different architectures, we only considered a fixed number N of consecutive basic blocks, where N corresponds to the number of pipeline stages.

4 Results

The quality of WCET predictions for the Cortex M4 and Cortex M7 architectures is evaluated in Sections 4.1 and 4.2. The effect of the different features of CAWET on the quality of the predictions is studied in Section 4.3. Finally, CAWET is evaluated in Subsection 4.4 on a more complex processor, the Cortex-A53, using a software measurement method and an operating system, allowing us to compare the WCET predictions of CAWET with those of WE-HML [2].

4.1 Quality of WCET predictions for the Cortex M4

Table 4 compares the WCET predictions of the selected TacleBench programs on the deterministic cache-less architecture Cortex M4. WCET predictions of BBs are either obtained by CAWET or by the context-agnostic Neural Network (NN) baseline described in Section 3.2. The table gives for the two techniques both the WCET prediction in cycles and the Relative Percentage Error RPE defined as $RPE = \frac{(Predict - Actual)}{Actual} * 100$. A context size of 3 BB is used.

The results show that CAWET is twice less pessimistic than the NN baseline on average, using the Mean Absolute Error[6] on the RPE (i.e., Error = RPE). This can be explained by the fact that: (i) neural networks do not consider the ordering of instructions in BBs (ii) neural networks are context-agnostic. We also observe that neither CAWET nor the NN baseline underestimates the WCET since all RPE are positive.

[6] Mean Absolute Error: MAE $= \frac{1}{n} * \sum^{n} |Error|$

Table 4 Comparison of WCET predictions for CAWET and a Neural Network (NN) baseline on TacleBench programs for Cortex-M4.

Benchmark	Maximum observed execution time (Cycles)	NN estimations (Cycles)	NN RPE (%)	CAWET estimations (Cycles)	CAWET RPE (%)
bs	140	307	119.2	272	94.3
bsort	317279	414882	30.7	374712	18.1
countnegative	9638	14047	45.7	12858	33.4
crc	78496	102005	29.9	92872	18.3
expint	5683	7758	36.5	5727	0.7
fdct	7308	10557	44.4	8606	17.7
fir	6882	10844	57.5	7490	8.8
h264_dec	573752	661037	15.2	607918	5.9
insertsort	3125	3964	26.8	3898	24.7
jfdctint	7761	11454	47.5	9968	28.4
matrix1	440243	577831	31.2	564921	28.3
ns	28444	45026	58.2	34367	20.8
petrinet	3283	4159	26.7	3592	9.4
Avg. MAE	–	–	43.80	–	**23.8**

Impact of the context size. Table 5 shows the considered context size's impact on the prediction quality. Four values are considered: 0 (no context), 1 BB as context, 3 BBs as context, and 20 BBs as context.

Table 5 Impact of the context size on the Mean Absolute Error (MAE) on TacleBench programs for Cortex-M4.

Benchmark	Context 0	Context 1 BB	Context Pipeline size (3)	Context 20 BB
bs	104,2%	97,6%	94,3%	117,9%
bsort	22,4%	27,6%	18,1%	34,2%
countnegative	47,3%	38,9%	33,4%	46,2%
crc	19,6%	11,1%	18,3%	19,3%
expint	21%	15,9%	0,7%	21,6%
fdct	39,2%	28,4%	17,7%	38,2%
fir	34,5%	31,6%	8,8%	39%
h264_dec	30,2%	22,1%	5,9%	30,9%
insertsort	15,5%	25,6%	24,7%	27,4%
jfdctint	34,6%	31,9%	28,4%	41,9%
matrix1	36,1%	33,3%	28,3%	53,4%
ns	45,7%	33,8%	20,8%	41,3%
petrinet	11%	17,2%	9,4%	16%
Avg. MAE	35,5%	31,9%	**23,8**%	40,6%

The results show that, on average, the error is minimal when the context size is 3 BBs. Accounting for the execution context of BBs is beneficial to the quality of the predictions up to a context size of 3. Taking into account larger context sizes results in much higher error values. One possible explanation for these higher error values is that the context vector is being disrupted by extensive information that cannot be processed efficiently with the current TXL architecture. In future works, we plan to examine this phenomenon more closely, which will require substantial computing resources.

4.2 Quality of WCET predictions for the Cortex M7

The Cortex M7 processor is more complex than the Cortex M4. It features a 6-stage in-order pipeline, data, and instruction caches with random cache replacement and a branch predictor. Table 6 evaluates WCET predictions produced by CAWET and the baseline NN for the Cortex M7, using a context size of 6 for CAWET.

Table 6 Comparison of WCET predictions for CAWET (vanilla) and a Neural Network (NN) baseline for Cortex-M7.

Benchmark	Maximum observed execution time (Cycles)	NN estimations (Cycles)	NN RPE (%)	CAWET estimations (Cycles)	CAWET RPE (%)
bs	140	307	119.3	280	100.0
bsort	191406	464616	142.7	376784	96.9
countnegative	6956	15874	128.2	13904	99.9
crc	47476	98473	107.4	88668	86.8
expint	3592	8260	130.0	7140	98.8
fdct	4957	12044	143.0	9341	88.4
fir	4625	10856	134.7	9132	97.4
h264_dec	362349	779905	115.2	706162	94.9
insertsort	1760	4188	138.0	3414	94.0
jfdctint	4011	11877	196.1	10215	154.7
matrix1	301866	660739	118.9	644668	113.6
ns	21253	46004	116.5	41167	93.7
petrinet	1595	3741	134.5	3342	109.5
Avg. MAE	–	–	132.7	–	**102.2**

The results show that even with no explicit support for caches, CAWET never underestimates compared to the Maximum observed execution time (the max of 1000 executions) and is again more precise than the NN baseline. It should also be noted that the average MAE, both for CAWET and NN, is, as one would expect, higher for the more complex Cortex M7 than for the very simple Cortex M4, showing that the tight timing analysis of complex processors is harder to achieve than the analysis of simpler ones.

Since the context size in CAWET is limited, the reuse of code/data (with instruction/data caches) may not be fully taken into account by the model. We thus modified CAWET to add a cache miss penalty to the WCET of a BB when the static cache analysis of Heptane cannot guarantee a cache hit. The same procedure is applied to the NN baseline, and the results are reported in Table 7.

The integration of cache analysis results into CAWET and NN leads to more pessimistic WCETs for both techniques. Two factors explain this additional pessimism: (i) the static cache analysis for random cache replacement is inherently pessimistic; (ii) CAWET already captures parts of the cache behavior due to its use of the execution contexts for BBs. Thus the impact of some cache misses may be counted twice.

4.3 Impact of CAWET features (Cortex M4 and M7)

In this section, we analyze the effect of different features of CAWET on the Relative Percentage Error (RPE): context accounting, peek-on mechanism, loop management, and using Heptane's cache analysis. Our study involves a comparison of the impact of each feature, starting with context accounting (A), followed by the peek-on mechanism (B), loop

Table 7 Comparison of WCET predictions for CAWET and a Neural Network (NN) baseline for Cortex-M7 when accounting for the static cache analysis results.

Benchmark	Maximum observed execution time (Cycles)	NN estimations (Cycles)	NN RPE (%)	CAWET estimations (Cycles)	CAWET RPE (%)
bs	140	537	283.6	516	268.6
bsort	191406	840959	339.4	699961	265.7
countnegative	6956	33552	382.3	26997	288.1
crc	47476	184152	287.9	166025	249.7
expint	3592	14528	304.5	12764	255.3
fdct	4957	34861	603.3	20076	305.0
fir	4625	18088	291.1	16554	257.9
h264_dec	362349	1281479	253.7	1403042	287.2
insertsort	1760	6040	243.2	7105	303.7
jfdctint	4011	34044	748.8	19663	390.2
matrix1	301866	2021791	569.8	1249975	314.1
ns	21253	97870	360.5	76205	258.6
petrinet	1595	5813	264.5	6372	299.5
Avg. MAE	–	–	398.1	–	**289.6**

unrolling (C), and finally, applying cache analysis (D). The results in Table 8 show that incorporating the context (A) provides the most significant improvement to CAWET, while the effects of peeking (B) and loop enrolling (C) are less substantial. Additionally, we can see that adding the cache analysis (D) in Cortex M7 has a considerable impact on the predictions, with a significant increase in pessimism.

Table 8 RPE measures of CAWET predictions for Cortex-M4 and Cortex-M7 when adding different features of CAWET: context accounting (A), peek-on mechanism (B), loop unrolling (C), and cache analysis (D).

Feature(s) \Optimization	Cortex-M4 RPE (%)	Cortex-M7 RPE (%)
None	35.5	142.5
A	25.2	130.2
A+B	24.9	126.1
A+B+C	**23.8**	**102.2**
A+B+C+D	NA	288.0

4.4 Quality of WCET predictions for the Cortex A53

The objectives of these experiments are twofold: (i) evaluate the WCET predictions produced by CAWET for a more complex processor than the Cortex M7; (ii) be able to compare CAWET to WE-HML [2], the related work closest to CAWET, that targets this architecture. We re-use the very same experimental conditions as in WE-HML: software measurements of execution times, and execution on top of an operating system. The maximum measured BB execution time is used alongside its context to train CAWET. We have collected 1000 measurements for each studied benchmark and kept the maximum execution time observed as a reference value to calculate the RPE. On the thousand measurements collected, we have also applied the probabilistic WCET technique as described in [30], where we set the probability to 10^{-3} to provide another reference point than the MOET.

Table 9 Comparison of WCET predictions on Cortex A53 for: CAWET, a probabilistic WCET solution, WE-HML, CAWET (vanilla), and a modified CAWET to account for static cache analysis results.

Benchmark	MOET (Cycles)	pWCET10^{-3} RPE (%)	WE-HML RPE (%)	Vanilla CAWET RPE (%)	CAWET with cache analysis RPE (%)
bs	2568	43.8	177.1	97.0	122.8
bsort	358380	60.4	838.3	18.6	21.3
countnegative	29720	6.3	168.5	70.2	169.6
crc	66867	64.2	315.2	53.8	86.5
expint	6122	1.0	352.5	29.0	80.3
fdct	8877	1.2	195.0	25.5	52.2
fir	7646	-13.6	391.4	31.1	114.9
h264_dec	426327	120.4	590.0	76.5	88.4
insertsort	3042	75.8	297.6	29.6	40.2
jfdctint	8070	51.1	296.1	44.4	57.5
matrixl	21380	5.8	207.1	223.9	236.6
ns	22018	-0.3	731.1	108.6	119.5
petrinet	3920	30.7	1865.3	2.3	30.8
Avg. MAE	–	36.5	494.2	62.4	93

Table 9 shows the Maximum Observed Execution Times (MOET) and Relative Percentage Error (RPE) for all considered techniques: probabilistic WCET estimation, WE-HML, Vanilla CAWET, and CAWET modified with the results of static cache analysis. On all benchmarks but one (matrix1), CAWET is much less pessimistic than WE-HML (even for the modified CAWET). This is due to the significant pessimism introduced by WE-HML to account for caches (WE-HML evaluates cache effects by generating the worst possible cache pollution in loops regardless of the actual accesses performed in the loop).

Compared to the probabilistic technique, we observe that the pWCET is sometimes unsafe. This may come from rare outliers (due, for example, to the presence of an operating system) that are considered as WCET and that pWCET (smartly) ignores because they are sufficiently rare. It may also happen when pWCET is less pessimistic than CAWET. However, in general, pWCET techniques may miss the worst-case execution path in programs, whereas CAWET, a hybrid technique, will not.

5 Related works

The challenge of accurately estimating the WCET of programs has led to the development of various hybrid timing analysis techniques that are compared with CAWET below. These techniques can be broadly categorized into two types: those that use measurements to estimate the WCET of individual basic blocks and those that incorporate machine learning to learn the BB's timing patterns.

1. **Hybrid WCET estimation techniques using measurements**
   AbsInt [11, 20, 21] and Rapita [4, 32] have developed hybrid WCET estimation solutions, namely *Timeweaver* and *Rapitime*, which rely on hardware-assisted measurements (e.g., JTAG) and manual annotations (way/trace points and interest points, respectively) to measure the WCET on code snippets, and then estimate the WCET of the program with

their static tool. Kirner et al. propose in [22] to perform measurements on code segments larger than a basic block and propose techniques to enforce coverage of the measured segments. In contrast to these research works, CAWET does not use measurements to estimate the WCET of code snippets. Instead, it utilizes a timing model learned through Machine Learning (ML) techniques.

2. **Hybrid WCET estimation techniques using ML**
   Several methods for estimating Worst-Case Execution Time (WCET) using Machine Learning (ML) have been proposed [5, 18, 17, 2, 24, 25]. Bonenfant et al. [5] use worst-case event counts for training a neural network, that will be subsequently used to calculate the WCET of a program at an early stage. Similarly, the approaches proposed by Kumar [24, 25] estimate WCET using features extracted from the source code. These approaches disregard valuable information about the code flow and hide the compilation effects by operating at the source code or intermediate code level, which can bias the timing prediction. The research works presented in [17, 2], similarly to CAWET, propose to extract features from the binary code and to use ML techniques to predict the WCET of individual basic blocks. However, contrary to [18, 17, 2], CAWET takes a more fine-grained approach, considering the context surrounding each basic block, and the dependencies between instructions within it to better consider hardware components such as the pipeline. [2] accounts for data caches by simulating the worst possible data access pattern for basic blocks within loops, whereas CAWET relies on static analysis through the Heptane tool [16] to obtain less pessimistic estimations of data cache behavior. [26] propose a technique similar to linear regression to estimate the WCET from a set of end-to-end measurements. Unlike CAWET, this approach uses static features and is thus not able to accurately predict pipeline effects.
   All approaches described in this section oversimplify the code characterization, either by using high-level abstractions of the source code or by relying on static features of basic blocks at the binary code level. In contrast, *CAWET* operates on the flow of instructions using state-of-the-art ML techniques (Transformers-XL).
3. **Machine Learning for contention prediction and throughput prediction**
   Brando et al. [6] use neural networks to estimate the worst contention factor of programs using hardware event counters. Similarly, Courtaud et al. [7] introduce a profiling tool that produces high-resolution profiles of the memory behavior of applications. They train a regressor using microbenchmarks to finally calculate contention. Even though these two studies rely heavily on ML, they focus on contention prediction on multi-core targets and not on WCET prediction for single-cores like CAWET.
   Deep PM [33] and Ithemal [27] employ transformers and LSTMs, respectively, to predict the throughput of isolated basic blocks. However, CAWET takes a different approach by incorporating the execution context to predict WCETs. Similarly, CATREEN [1] uses stacked LSTMs to forecast the average execution time of basic blocks in a contextualized manner, but it differs from CAWET in its focus on average execution time rather than worst-case execution time.

6 Conclusion

In this paper, we presented CAWET: a hybrid approach that estimates the worst-case program timing for individual basic blocks in a program. Our approach uses static techniques to identify the longest execution path and an advanced machine learning architecture called transformer-XL to predict the worst-case execution time of each basic block. By considering

the execution context formed by previously executed basic blocks, CAWET is able to account for the micro-architecture of the processor pipeline without explicit modeling. The technique is demonstrated to be empirically reliable and less pessimistic than its competitors in experiments on the TacleBench benchmarks for different target processors. While there are still challenges to be addressed, such as the need for more accurate context for less pessimistic predictions, CAWET offers a promising solution for predicting worst-case execution times for Commercial off-the-shelf processors. In future work, the technique will be further explored for processors with out-of-order pipelines, such as Cortex A9 or A72.

References

1 Abderaouf N. Amalou, Elisa Fromont, and Isabelle Puaut. Catreen: Context-aware code timing estimation with stacked recurrent networks. In *34rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI)*. IEEE, 2022.

2 Abderaouf N Amalou, Isabelle Puaut, and Gilles Muller. We-hml: hybrid wcet estimation using machine learning for architectures with caches. In *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 31–40. IEEE, 2021.

3 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: 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings 8*, pages 35–46. Springer, 2010.

4 Adam Betts, Nicholas Merriam, and Guillem Bernat. Hybrid measurement-based wcet analysis at the source level using object-level traces. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2010.

5 Armelle Bonenfant, Denis Claraz, Marianne De Michiel, and Pascal Sotin. Early wcet prediction using machine learning. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, pages 5–1. OASICs, Dagstuhl Publishing, 2017.

6 Axel Brando, Isabel Serra, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. Using quantile regression in neural networks for contention prediction in multicore processors. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

7 Cédric Courtaud, Julien Sopena, Gilles Muller, and Daniel Gracia Pérez. Improving prediction accuracy of memory interferences for multicore platforms. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 246–259. IEEE, 2019.

8 Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint*, 2019. `arXiv:1901.02860`.

9 Jean-François Deverge and Isabelle Puaut. Safe measurement-based wcet estimation. In *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2007.

10 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, pages 4171–4186. Association for Computational Linguistics, 2019.

11 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous non-intrusive hybrid wcet estimation using waypoint graphs. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016.

12 Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. Cyclomatic complexity. *IEEE software*, 33(6):27–29, 2016.

13 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

14 SEGGER Microcontroller GmbH. Ozone User Guide & Reference Manual. URL: `https://www.segger.com/`.

15 Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *4th IEEE international workshop on workload characterization*, 2001.

16 Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017.

17 Thomas Huybrechts, Thomas Cassimon, Siegfried Mercelis, and Peter Hellinckx. Introduction of deep neural network in hybrid wcet analysis. In *Advances on P2P, Parallel, Grid, Cloud and Internet Computing: Proceedings of the 13th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2018)*, pages 415–425. Springer, 2019.

18 Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx. A new hybrid approach on wcet analysis for real-time systems using machine learning. In *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2018.

19 Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa, editors, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pages 171–185. ACM, 1994. `doi:10.1145/178243.178258`.

20 Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. Obtaining worst-case execution time bounds on modern microprocessors. In *Embedded World Conference*, 2018.

21 Daniel Kästner, Markus Pister, Simon Wegener, and Christian Ferdinand. Timeweaver: A tool for hybrid worst-case execution time analysis. In *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2019.

22 Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. Using measurements as a complement to static worst-case execution time analysis. *Intelligent Systems at the Service of Mankind*, 2(8):20, 2005.

23 Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint*, 2018. `arXiv:1808.06226`.

24 Vikash Kumar. Deep neural network approach to estimate early worst-case execution time. In *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, pages 1–8. IEEE, 2021.

25 Vikash Kumar. Estimation of an early wcet using different machine learning approaches. In *Advances on P2P, Parallel, Grid, Cloud and Internet Computing: Proceedings of the 17th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC-2022)*, pages 297–307. Springer, 2022.

26 Björn Lisper and Marcelo Santos. Model identification for wcet analysis. In *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 55–64. IEEE, 2009.

27 Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*, pages 4505–4515. PMLR, 2019.

28 Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint*, 2021. `arXiv:2105.12655`.

29 Federico Reghenzani, Giuseppe Massari, William Fornaciari, and Andrea Galimberti. Probabilistic-wcet reliability: On the experimental validation of evt hypotheses. In *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, pages 229–234, 2019.

30 Federico Reghenzani, Luca Santinelli, and William Fornaciari. Dealing with uncertainty in pwcet estimations. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(5):1–23, 2020.

31 Segger. J-Trace PRO – The Leading Trace Solution. URL: `https://www.segger.com/products/debug-probes/j-trace/`.

32 Hardik Shah, Andrew Coombes, Andreas Raabe, Kai Huang, and Alois Knoll. Measurement based wcet analysis for multi-core architectures. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, pages 257–266, 2014.

33 Jun S Shim, Bogyeong Han, Yeseong Kim, and Jihong Kim. Deeppm: transformer-based power and performance prediction for energy-aware software. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1491–1496. IEEE, 2022.

34 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

35 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

36 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

37 Tomofumi Yuki. Understanding polybench/c 3.2 kernels. In *International workshop on polyhedral compilation techniques (IMPACT)*, pages 1–5, 2014.

A Appendix

Table 10 Hyperparameters used during transformer XL pretraining and finetuning.

Hyperparameters	Pretraining phase	Finetuning phase
Number of layer	4	4
Number of attention heads	3	3
Dimension of head	22	22
Dimension of inner head	128	128
Dimension of hidden layers	512	512
Optimizer	“adam”	“adam”
Target length	512	512
Memory length	1024	1024
Linear layer (fine tuning)	–	{512, 256, 128, 1}
Learning rate	0.00025	0.0001

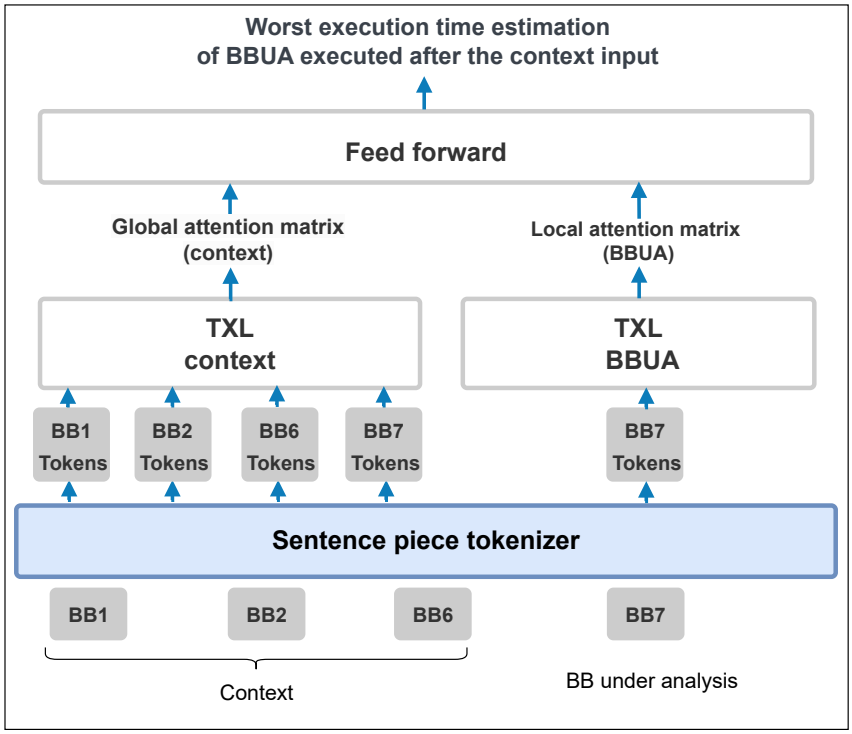

Figure 4 CAWET Transformer XL system architecture.