Machine learning for timing estimation

Abderaouf Nassim Amalou

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par

« Abderaouf Nassim AMALOU »

« Machine Learning for timing estimation »

Thèse présentée et soutenue à « Rennes », le « 12 décembre 2023» (date préliminaire)
Unité de recherche : « Univ. Rennes, INRIA, CNRS, IRISA»

Rapporteurs avant soutenance :

Claire PAGETTI — Ingénieure de Recherche (HDR), ONERA Toulouse
Jalil BOUKHOBZA — Professeur, ENSTA Bretagne

Composition du Jury :

Président :		
Examinateurs :	Olivier SENTIEYS	Professeur, Université de Rennes
	Smail NIAR	Professeur, Université Polytechnique Hauts-de-France
	Claire PAGETTI	Ingénieure de Recherche, ONERA Toulouse
	Jalil BOUKHOBZA	Professeur, ENSTA Bretagne
Dir. de thèse :	Isabelle PUAUT	Professeure, Université de Rennes
Co-dir. de thèse :	Elisa FROMONT	Professeure, Université de Rennes

Acknowledgement

En préambule de ce manuscrit de thèse, je souhaiterais adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et ont ainsi contribué à l'élaboration de ce modeste travail.

Je tiens à exprimer ma profonde gratitude envers mes encadrantes de thèse Pr. Isabelle PUAUT et Pr. Elisa FROMONT, enseignantes-chercheuses à l'université de Rennes, qui m'ont marqué au cours de ces trois années par leurs précieux conseils, leur écoute, leur réactivité et leur disponibilité. La confiance qu'elles m'ont accordée et leur sens de la pédagogie ont affirmé ma détermination et m'ont permis de mener à terme ce travail.

Je souhaiterais adresser mes profonds remerciements aux membres du jury qui m'ont fait l'honneur d'accepter de juger ce travail.

Je suis particulièrement reconnaissant envers les membres de l'équipe PACAP : Erven, Pierre, Damien, Caroline, Anis, Camille, Pierre, Nicolas, Hugo, Antoine, Sara, Nicolas, Aurore, Hector, ainsi que les membres de l'équipe LACADOM : Alexandre, Tassadit, Sébastien, Luis, Louis, Christine, Laurance, Véronique, Peggy, Romaric, Julien, Camille, Lénaïgue, pour tous les bons moments que nous avons passés ensemble.

J'aimerais adresser un remerciement particulier à Virginie Desroches et Gaelle Tworkowski pour leur disponibilité, leur amabilité et leur gentillesse.

Je ne saurais terminer sans exprimer mes remerciements les plus chaleureux à toute ma famille, en particulier à ma mère et mon père, à ma sœur, à mon grand frère ainsi qu'à ma belle-sœur, pour le soutien et les encouragements qu'ils m'ont apportés avec un dévouement total.

Pour finir, je remercie tous mes amis et en particulier ma femme Manele, dont la bienveillance et l'encouragement ininterrompus m'ont permis de mener à bien ce travail.

TABLE OF CONTENTS

List of Figures

List of Tables

Résumé

Les systèmes embarqués sont des dispositifs électroniques contrôlés par un logiciel pour effectuer des tâches spécifiques. Ces tâches vont de l'exploitation d'appareils programmables à la maison à la gestion des systèmes dans les voitures et les avions. La prévalence de ces systèmes a considérablement augmenté, comme en témoignent les projections suggérant que le nombre de dispositifs Internet des objets (IoT) atteindra 50 milliards d'ici 2030 [4]. Au fur et à mesure que les nouvelles technologies évoluent, elles présenteront inévitablement de nouveaux défis liés à la taille, au coût et aux performances de ces systèmes embarqués. Par conséquent, les concepteurs de systèmes doivent comprendre le comportement du logiciel embarqué en ce qui concerne ces contraintes. Cependant, la complexité croissante des architectures matérielles, associée à une documentation insuffisante, complique la tâche d'estimer les performances des logiciels. Les performances peuvent être des performances de pire en pire (WCETs) dans les systèmes en temps réel ou les performances moyennes de cas dans ceux à usage général.

Dans les systèmes à usage général, l'utilisation efficace des ressources est essentielle. L'une des façons d'améliorer les mesures de rendement, comme le temps d'exécution moyen, est par le biais de transformations ou d'optimisations au niveau du code. Toutefois, l'évaluation précise de ces optimisations nécessite la compréhension de divers facteurs, y compris l'interaction entre les instructions du programme. Pour faciliter ce processus d'évaluation complexe, des outils spécialisés ont été conçus pour quantifier le temps d'exécution, en se concentrant spécifiquement sur l'impact des optimisations. Selon les ressources disponibles, ces outils utilisent généralement soit des techniques de profilage [135, 49] lorsque le matériel cible est accessible, soit des simulateurs de processeurs [19, 13, 3, 120], qui sont des outils logiciels qui émulent le comportement d'un processeur, permettant ainsi l'analyse des performances et les tests de logiciel sans avoir besoin du matériel réel.

Dans les systèmes en temps réel, les tâches viennent avec des délais spécifiques qui doivent être respectés pour considérer le système comme fonctionnant correctement. S'assurer que les tâches sont terminées à temps nécessite l'évaluation du WCET pour chaque tâche. Cette évaluation aide l'algorithme de planification à allouer des ressources afin que chaque tâche atteigne sa date limite, même dans les scénarios les plus exigeants. Pour une estimation précise de WCET, différentes méthodes peuvent être employées, qui impliquent généralement une considération simultanée du code de la tâche et de l'architecture du

processeur qui l'exécute.

Les méthodes d'estimation WCET sont divisées en méthodes *static*, méthodes *basées sur la mesure de l'end-to-end* et méthodes hybrides *hybrid* [170]. *Static methods* évalue le WCET sans exécuter le programme. Dans la première phase, le programme est divisé en blocs de base. Un bloc de base (BB) est une séquence d'instructions avec un point d'entrée unique et un point de sortie unique. Le WCET de chaque BB est estimé grâce à la connaissance de l'architecture du processeur. Dans la deuxième phase, les techniques statiques calculent l'estimation WCET pour l'ensemble du programme en fonction de la WCET de chaque BB dans le code. Pour cette deuxième phase, *Implicit Path Enumeration Technique* (IPET) [170, 117] est la classe de techniques la plus couramment utilisée. IPET s'appuie sur la résolution d'un problème d'optimisation linéaire généré à partir du graphique de flux de contrôle du programme (CFG). Les méthodes statiques fournissent une estimation *safe* WCET, qui est une limite supérieure de tout temps d'exécution possible, à condition que l'estimation WCET de chaque bloc de base soit elle-même sûre.

End-to-end measurement-based méthodes sont des techniques empiriques qui ne nécessitent pas de connaissances détaillées du matériel. Ils lancent le programme sur une série d'entrées, et les temps d'exécution résultants sont mesurés et recueillis. Le WCET est ensuite estimé, soit en considérant le WCET comme la mesure la plus élevée, soit par extrapolation en utilisant des techniques statistiques [30]. Par construction, lorsque l'on utilise la mesure la plus élevée comme estimation WCET, ces techniques ne peuvent que sous-estimer la WCET à moins que l'entrée et l'état matériel résultant du chemin d'exécution le plus long ne soient utilisés lors des tests [51]. Par conséquent, une marge de sécurité est souvent ajoutée à l'estimation WCET pour atténuer le manque de confiance dans les mesures.

Les méthodes *Hybrid* mélangent des approches statiques et basées sur la mesure. Dans une grande majorité de ces techniques (par exemple, [**BETT:06 a**, 30, 157, 59]), *mesures* sont utilisées pour estimer le WCET des blocs de base. Le WCET de l'ensemble du programme est ensuite estimé en utilisant des méthodes de calcul telles que l'IPET. L'avantage des techniques hybrides est qu'elles ne nécessitent pas de connaissance de l'architecture tout en étant en mesure de trouver le chemin le plus long.

Néanmoins, les outils actuels pour l'estimation WCET, qu'ils soient statiques, end-to-end ou hybrides, présentent chacun leur propre ensemble de défis. Les outils d'analyse statique nécessitent une compréhension approfondie de la microarchitecture du processeur, y compris des aspects tels que les caches [67], les pipelines [113] et les prédicteurs de branches citeCOLI:00a. L'acquisition d'une telle connaissance détaillée des microarchitectures devient de plus en plus difficile, soit en raison de restrictions de propriété intellectuelle, soit parce que la complexité des architectures modernes complique l'élaboration

de modèles de timing fiables et sûrs. Les méthodes de bout en bout utilisent soit des outils de profilage, soit des simulateurs de cycle précis. Alors que les outils de profilage peuvent perturber la mesure des performances pendant leur fonctionnement, les simulateurs cycle précis, bien que précis, sont intrinsèquement riches en ressources et peuvent être lents. Les deux approches dans le cadre des méthodes end-to-end peuvent également manquer de caractéristiques de sécurité cruciales. Pendant ce temps, les techniques hybrides souffrent de problèmes tels que la couverture complète du code [1] [111]. Compte tenu de ces défis, il y a un besoin urgent de méthodes plus simples et plus efficaces pour la modélisation du temps complexe des processeurs.

Au cours de la dernière décennie, le Machine Learning (ML) est rapidement devenu un outil révolutionnaire dans de nombreux domaines, du secteur des véhicules autonomes aux diagnostics améliorés dans les soins de santé. De même, le rôle de ML dans l'architecture informatique a évolué d'un concept théorique à une technologie fondamentale, influençant la conception, le contrôle et la simulation de divers composants du système. Historiquement, l'interaction entre ML et l'architecture informatique s'est largement concentrée sur l'adaptation d'éléments architecturaux pour mieux servir les algorithmes ML tels que les accélérateurs de réseaux neuronaux [36]. Cependant, la dernière décennie a marqué un changement significatif vers une relation plus réciproque, car de plus en plus de travaux appliquent avec succès ML à l'architecture de processeur et à la résolution de problèmes de conception de compilateur [**mlsYS_survey2018**, 32].

Compte tenu des défis croissants associés à la complexité du matériel et à la documentation limitée, cette thèse vise à **automatiser la création de modèles de timing matériel**. En tirant parti des techniques d'apprentissage automatique, l'objectif est de prédire les performances des cas moyens et des cas les plus mauvais sans nécessiter une documentation approfondie du processeur ciblé. Les solutions proposées fonctionnent toutes en deux phases distinctes. Au cours de la phase d'apprentissage *learning phase*, le timing des fragments de code est établi sur la base des mesures dans leurs divers "contexts d'exécution", y compris les boucles et les dépendances entre les instructions. Dans la phase *inferring* ultérieure, le modèle de timing développé est appliqué pour calculer le temps d'exécution de nouveaux fragments de code. Ces calculs sont censés être informés par le contexte d'exécution de chaque fragment de code, déterminé par exemple par une analyse statique. Cette méthodologie offre trois avantages clés :

— Il fournit des estimations de temps raisonnablement précises et rapides.

— Il élimine la nécessité d'analyses statiques coûteuses ou de simulations cycle-exactes.

— Il ne nécessite aucune connaissance détaillée de la microarchitecture du processeur.

1. Assurer une couverture complète du code signifie vérifier que chaque partie du code du logiciel a été exécutée et analysée, ne laissant aucune section non testée ou non vérifiée.

Bien que l'information de timing obtenue ne soit pas sûre, elle a une valeur significative pour l'estimation du timing dans les premiers stades du développement du système, des systèmes en temps réel à faibles niveaux de critique, des logiciels à usage général, ou des optimisations de compilateur de guidage.

Alors que des recherches antérieures ont été menées sur la dérivation automatique de modèles de timing en utilisant l'apprentissage automatique [21, 87], elle s'est concentrée principalement sur le matériel simple avec un timing d'instruction constant et indépendant du contexte. Cette thèse vise à ouvrir un nouveau terrain en introduisant la prise de conscience du contexte dans ces techniques d'apprentissage automatique, étendant ainsi leur applicabilité à des conceptions de matériel plus complexes. Spécifiquement, les traces d'exécution d'un programme serviront de représentations contextuelles pour les séquences d'instructions pour lesquelles il faudra estimer les temps de l'execution. En tirant une analogie du domaine du traitement automatique du langage naturel (TAL), ces traces d'exécution peuvent être considérées comme des textes dans lesquels des instructions individuelles peuvent être vues comme des mots. En traitant ces traces, nous pouvons acquérir des connaissances précieuses sur les facteurs contextuels qui influent sur le temps d'exécution des instructions (effets de pipeline, effets de cache et effets prédicteurs de branches), en tirant parti des progrès en TAL pour guider nos modèles. Cette nouvelle approche promet d'améliorer l'exactitude des estimations des délais d'exécution des programmes, ce qui peut à son tour conduire au développement de systèmes intégrés plus efficaces et fiables.

Contribution

Cette thèse introduit de nouvelles méthodologies à l'intersection entre l'apprentissage automatique et l'estimation du temps d'exécution. Les trois contributions de cette thèse sont les suivantes :

estimation WCET hybride à l'aide de l'apprentissage automatique pour les architectures avec caches [7]. Notre proposition initiale est une nouvelle approche hybride, WE-HML [7], conçue pour améliorer l'estimation WCET. Cette méthode intègre de manière distincte les considérations de la mémoire cache de données lors de la formation d'une gamme de modèles d'apprentissage automatique fondamentaux. La formation utilise des ensembles de données synthétiquement générés et est complétée par une technique statique pour estimer le WCET global d'un programme.

Utilisation de la TAL dans l'estimation ACET. En nous tournant vers ACET, nous explorons l'intégration des techniques de TAL pour capturer les dépendances entre les séquences d'instruction. Nous enquêtons sur diverses architectures d'apprentissage

profond, y compris la mémoire à court terme [82] (comme publié dans le journal [8]) et Transformers [56, 43]. Nous avons constaté que Transformers XL [43] était le mieux adapté pour contextualiser et estimer avec précision les temps d'exécution de blocs de base.

Évaluation WCET en connaissance de contexte à l'aide de Transformers [6]. Sur la base du succès de l'application TAL dans l'estimation ACET, nous prenons le défi d'identifier le contexte du pire des cas pour les blocs de base, conduisant à la conception de CAWET [6]. Cette nouvelle solution identifie non seulement chaque contexte d'exécution court pour un bloc de base donné, mais elle exploite également Transformers XL [43] pour améliorer la précision de l'estimation WCET. En outre, nous avons intégré ces améliorations dans un outil d'analyse statique, créant une méthodologie hybride qui atténue considérablement les surestimations observées dans notre modèle initial WE-HML.

Outline

Le reste de ce document est organisé comme suit :

Chapitre 1. Nous mettons les bases en présentant les concepts clés nécessaires à la compréhension de ce document. Nous examinons les méthodes courantes pour estimer les délais d'exécution dans les scénarios généraux et examinons des techniques spécialisées pour l'estimation WCET. Ce chapitre sert également d'introduction à certaines techniques de régression et d'apprentissage profond. Le chapitre se termine par un aperçu des applications de pointe de l'apprentissage automatique pour l'estimation du temps d'exécution.

Chapitre 2. Ce chapitre a été le premier travail achevé au cours de cette thèse. Il présente une nouvelle méthodologie hybride pour l'analyse WCET en utilisant des techniques de base d'apprentissage automatique. Dans ce chapitre, nous accordons une attention particulière aux effets de mémoire, plus spécifiquement, aux comportements de cache qui peuvent se produire dans des boucles ancrées lors de l'exécution de code.

Chapitre 3. Dans ce chapitre, nous nous aventurons dans le domaine du traitement automatique du langage naturel (TAL), en explorant des techniques avancées d'apprentissage automatique telles que LSTM et Transformers [165] pour estimer les temps d'exécution moyens de cas de fragments de code. Nous accordons ici une attention particulière au rôle du contexte d'exécution dans ces estimations.

Chapitre 4. Nous revenons à l'estimation WCET, où nous nous appuyons sur les conclusions des chapitres précédents en améliorant le modèle hybride du Chapitre 2 en utilisant les modèles les plus efficaces identifiés dans le Chapitres refch3. Le défi de ce chapitre est d'identifier le contexte d'exécution dans le pire des cas, ce qui est crucial pour

appliquer efficacement ces modèles d'apprentissage automatique dans l'estimation WCET hybride.

Chapitre 5. Enfin, nous terminons la thèse avec un regard vers l'avenir, en discutant des travaux futurs et des voies d'amélioration.

INTRODUCTION

Embedded systems are electronic devices controlled by software to perform specific tasks. These tasks range from operating programmable appliances at home to managing systems in cars and airplanes. The prevalence of these systems has increased significantly, as evidenced by projections suggesting that the number of Internet of Things (IoT) devices will reach 50 billion by 2030 [4]. As new technologies evolve, they will inevitably introduce further challenges related to the size, cost, and performance of these embedded systems. Therefore, system designers need to understand the embedded software's behavior with respect to these constraints. However, the increasing complexity of hardware architectures, coupled with insufficient documentation, complicates the task of estimating software performance. Performance may be a worst-case performance (Worst-Case Execution Times - WCETs) in real-time systems or average-case performance in general-purpose ones.

In general-purpose systems, efficient utilization of resources is crucial. One way to enhance performance metrics, like average execution time, is through code-level transformations or optimizations. However, accurately evaluating these optimizations requires understanding various factors, including the interplay among program instructions. To facilitate this complex evaluation process, specialized tools have been designed to quantify execution time, specifically focusing on the impact of optimizations. Depending on the available resources, these tools typically employ either profiling techniques [135, 49] when the target hardware is accessible, or processor simulators [19, 13, 3, 120], which are software tools that emulate the behavior of a processor, thereby enabling performance analysis and software testing without the need for the actual hardware.

In real-time systems, tasks come with specific deadlines that need to be met to consider the system as functioning correctly. Ensuring tasks are completed on time requires evaluating the WCET for each task. This evaluation helps the scheduling algorithm allocate resources so that every task meets its deadline, even in the most demanding scenarios. For accurate WCET estimation, various methods can be employed, which typically involve a simultaneous consideration of both the task's code and the architecture of the processor executing it.

WCET estimation methods are divided into *static* methods, *end-to-end measurement-based* methods, and *hybrid* methods [170]. *Static methods* estimate the WCET without executing the program. In the first phase, the program is divided into basic blocks. A Basic Block (BB) is a sequence of instructions with a single entry point and a single exit

point. The WCET of each BB is estimated thanks to the knowledge of the processor architecture. In the second phase, static techniques calculate the WCET estimate for the whole program based on the WCET of each BB within the code. For this second phase, *Implicit Path Enumeration Technique* (IPET) [170, 117] is the most commonly 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.

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 considering the WCET as the highest measurement or by extrapolating using statistical techniques [30]. 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 [51]. 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., [100, 30, 17, 157, 59]), *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.

Nevertheless, current tools for WCET estimation, whether they are static, end-to-end, or hybrid, each come with their own set of challenges. Static analysis tools demand an in-depth understanding of the processor's microarchitecture, including aspects like caches [67], pipelines [113], and branch predictors [40]. Acquiring such detailed knowledge of microarchitectures is becoming more difficult, either due to intellectual property restrictions or because the complexity of modern architectures complicates the development of reliable and safe timing models. End-to-end methods utilize either profiling tools or cycle-accurate simulators. While profiling tools can disrupt performance measurement during their operation, cycle-accurate simulators, though precise, are inherently resource-intensive and can be slow. Both approaches within end-to-end methods may also lack crucial safety features. Meanwhile, hybrid techniques suffer from issues like ensuring complete code coverage [2] [111]. Given these challenges, there is a pressing need for more straightforward and efficient methods for complex processor timing modeling.

2. Ensuring complete code coverage means verifying that every part of the software code has been executed and analyzed, leaving no section untested or unchecked.

During the past decade, Machine Learning (ML) has quickly become a revolutionary tool in many fields, from the autonomous vehicles sector to enhanced diagnostics in healthcare [137, 106]. Similarly, ML's role in computer architecture has evolved from a theoretical concept to a foundational technology, influencing design, control, and simulation across various system components [163]. Historically, the interplay between ML and computer architecture largely focused on adapting architectural elements to better serve ML algorithms like neural network accelerators [36]. However, the last decade has marked a significant shift toward a more reciprocal relationship, as more and more works successfully apply ML to processor architecture and compiler design problem-solving [124, 169, 32].

Given the growing challenges associated with hardware complexity and limited documentation, this thesis aims at **automating the creation of hardware timing models**. By leveraging machine learning techniques, the objective is to predict both average-case and worst-case performance without requiring extensive documentation of the targeted processor. The proposed solutions all operate in two distinct phases. During the *learning phase*, the timing for code snippets is established based on measurements in their various "execution contexts", including loops and dependencies between instructions. In the subsequent *inferring phase*, the developed timing model is applied to calculate the execution time of new code snippets. These calculations are supposed to be informed by the execution context of each code snippet, as determined through, for example, static analysis. This methodology offers three key advantages:

— It provides reasonably accurate and fast timing estimations.

— It eliminates the need for costly static analyses or cycle-accurate simulation.

— It does not require detailed knowledge of the processor's microarchitecture.

Although the obtained timing information is not provably safe, it holds significant value for estimating timing in the early stages of system development, real-time systems at low criticality levels (for example, DAL B and C in the aeronautic industry [20]), general-purpose software, or guiding compiler optimizations.

While previous research has been conducted on the automatic derivation of timing models using machine learning [21, 87], it predominantly focused on simple hardware with constant and context-independent instruction timing. This thesis aims to break new ground by introducing context awareness into these machine learning techniques, thereby extending their applicability to more complex hardware designs. Specifically, execution traces of a program will serve as contextual representations for the instruction sequences for which execution times need to be estimated. Drawing an analogy from the field of Natural Language Processing (NLP), these execution traces can be thought of as texts in which individual instructions can be seen as words. By processing these traces, we can

gain valuable insights into the contextual factors that influence instruction execution time (pipeline effects, cache effects, and branch predictor effects), leveraging advancements in NLP to guide our models. This new approach promises to enhance the accuracy of program execution time estimates, which in turn can drive the development of more efficient and reliable embedded systems.

Contributions

This thesis introduces new methodologies at the crossroads of machine learning and execution time estimation. The three contributions of this thesis are as follows:

Hybrid WCET estimation using machine learning for architectures with caches [7]. Our initial proposition is a novel hybrid approach, WE-HML [7], designed for improved WCET estimation. This method distinctively incorporates data cache memory considerations when training a range of foundational machine learning models. The training utilizes synthetically generated datasets and is complemented with a static technique to estimate the overall WCET of a program.

Employment of NLP in ACET estimation. Shifting our focus toward ACET, we explore the integration of NLP techniques to capture the dependencies between instruction sequences. We investigate various deep learning architectures, including Long-Short Term Memory [82] (as published in the paper [8]) and Transformers [56, 43]. We found that Transformers XL [43] was the best suited for accurately contextualizing and estimating basic block execution times.

Context-aware WCET estimation using Transformers [6]. Building on the success of NLP application in ACET estimation, we take on the challenge of identifying "worst-case context" for basic blocks, leading to the conception of CAWET [6]. This novel solution not only identifies every short execution context for a given basic block, but it also leverages Transformers XL [43] to enhance WCET estimation accuracy. Moreover, we integrated these enhancements into a static analysis tool, creating a hybrid methodology that significantly mitigates the overestimations observed in our initial WE-HML model.

Outline

The rest of this document is organized as follows:

Chapter 1. We lay the groundwork by introducing the key concepts necessary for understanding this document. We review the prevalent methods for estimating execution times in general scenarios and delve into specialized techniques for WCET estimation. This

chapter also serves as a short introduction to some regression and deep learning techniques. The chapter concludes with an overview of cutting-edge applications of machine learning for execution time estimation.

Chapter 2. This chapter was the first work completed in the course of this thesis. It presents a novel hybrid methodology for WCET analysis using basic machine learning techniques. In this chapter, we pay close attention to memory effects, more specifically, to cache behaviors that may arise within nested loops during code execution.

Chapter 3. In this chapter, we venture into the realm of Natural Language Processing (NLP), exploring advanced machine learning techniques like LSTM and Transformers [165] to estimate average-case execution times of code snippets. Here, we give special emphasis to the role of execution context in these estimations.

Chapter 4. We return to WCET estimation, where we build upon the findings of the previous chapters by enhancing the hybrid model from Chapter 2 using the most effective models identified in Chapter 3. The challenge in this chapter is to pinpoint the worst-case execution context, which is crucial for applying these machine learning models effectively in hybrid WCET estimation.

Chapter 5. Finally, we conclude the thesis with a look ahead, discussing future work and avenues for further improvement.

Publications

Please note that for the following publications, the authors are arranged in order of their contribution, with the principal contributor listed first.

Abderaouf N., AMALOU, Isabelle PUAUT, and Gilles MULLER. "WE-HML: Hybrid WCET Estimation Using Machine Learning for Architectures with Caches." The 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). IEEE, 2021.

Abderaouf N., AMALOU, Elisa FROMONT, and Isabelle PUAUT. "CATREEN: Context-Aware Code Timing Estimation with Stacked Recurrent Networks." The 34th IEEE International Conference on Tools with Artificial Intelligence (ICTAI) IEEE, 2022.

Abderaouf N., AMALOU, Elisa FROMONT, and Isabelle PUAUT. "CAWET: Context-Aware Worst-Case Execution Time Estimation Using Transformers." The 35th Euromicro Conference on Real-Time Systems, (ECRTS 2023). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

In progress

Hugo REYMOND, **Abderaouf N., AMALOU**, Hector CHABOT and Isabelle PUAUT. "Worst-Case Execution Time and Energy Estimation in Low-Power Microprocessors using Explainable ML."

Abderaouf N., AMALOU, Elisa FROMONT, and Isabelle PUAUT. "Interpretable, Fast, and Accurate Context-Aware Basic Block Timing Prediction using Transformers."

Chapter 1

Background on Timing Estimation Using Machine Learning

Over the past decade, the application of machine learning techniques to estimate the execution time of programs has gained interest among researchers, yielding several solutions. Each method stands out for its distinct features, including its domain of use, the code level utilized to estimate its execution time, and the way timing is obtained. In this chapter, we dive into various classifications of these techniques, providing a comprehensive overview of their advantages and disadvantages. We conclude the chapter with an analysis of the current state-of-the-art in timing estimation using machine learning and outline potential future directions for research.

This chapter is organized as follows: first, a comprehensive overview of the context surrounding execution time estimation, in general, is provided in Section 1.1, encompassing different uses of execution time, factors that influence it, and the various techniques for its calculation or estimation. Subsequently, a focused investigation into Worst-Case Execution Time (WCET) estimation is conducted in Section 1.2. The utilization of machine learning techniques is introduced and further detailed in Section 1.3. The synergy between machine learning and the estimation of execution time is evaluated through a review of the existing literature in Section 1.4. Finally, Section 1.5 assesses the limitations in current state-of-the-art methods, setting the stage for the specific challenges and contributions addressed in this thesis.

1.1 Execution time estimation: a bird's-eye view

1.1.1 Levels of execution time estimation

Execution time is a usual metric for assessing the performance of a program or a system, and it can be understood at different levels.

Execution time of an instruction. At the lowest level, the execution time can refer to the time taken to execute a single instruction in a program. This could be a simple operation, such as an addition or a multiplication in a CPU. Microbenchmark tools can

be used for this purpose [2].

Execution time of a basic block (BB). A basic block is a code sequence with no branch *in*, except at the entry, and no branch *out*, except at the exit. In other words, if a single instruction is executed in the basic block, all the instructions are executed in sequence. Therefore, the execution time of a basic block would be the total time taken to execute all the instructions in the block. The basic block code level is widely used in realtime systems for worst-case execution time estimation and compiler design, where considering BB-level timing is a useful approach for understanding the performance characteristics of a program and guiding optimization efforts.

Execution time of a workload, function, program, or application. These terms are often used interchangeably to refer to larger units of execution. In each of these terms, the execution time would refer to the total time taken to complete all the operations, whether it is processing a workload, running a function, executing a program, or operating an application. This is typically a more complex measure, as it must account for various factors, such as function calls, control flow, resource usage, presence of multiple tasks, and interactions between software and hardware components.

This thesis's primary focus revolves around exploring program execution time. Specifically, the time analysis of the basic block, which not only serves to determine the worst-case execution time of programs in case of realtime applications, but also provides valuable insights that can be leveraged for compiler optimization techniques.

1.1.2 Execution time usages

Execution time estimation can be approached from different perspectives and categorized into average, best, and worst-case scenarios.

Average-case execution time (ACET). The average execution time is crucial for assessing the typical (mean or average) duration required to complete a task. It considers different inputs or different repeated execution scenarios to calculate the average. This metric is important for compiler optimization and microarchitecture design [149].

Best-case execution time (BCET). The best-case execution time serves as a reference point in performance analysis, defined as the minimum time necessary for a task to execute under optimal conditions. This metric is important for throughput analysis, aiding in the identification of potential system bottlenecks. A thorough understanding of the constraints and capabilities of the microarchitecture, obtained through BCET analysis, enables more precise finetuning of optimization strategies. Consequently, this enhances system throughput and overall efficiency [129].

Worst-case execution time (WCET). The worst-case execution time represents the maximum time required for a task or module to complete. It indicates the worst-case scenario and is crucial for determining timing constraints and ensuring system stability under extreme conditions. In safety-critical systems and realtime applications, it is essential to guarantee that the system meets its deadlines, even in worst-case scenarios. Consider the collision avoidance system of an autonomous vehicle. Estimating the worst execution time of the collision detection algorithm is vital to ensure that the system can respond within the required time frame to avoid accidents. By accounting for the worst-case execution time, designers can allocate sufficient processing resources and validate the system's ability to operate safely under all conditions (including the worst-case).

1.1.3 Factors behind the variability of execution times

Obtaining an accurate estimate of execution times is challenging due to interactions between hardware and software components. Understanding these interactions and how they affect the execution time variability is essential to improve the reliability and precision of timing estimation solutions. This Section focuses on explaining the key factors that contribute to timing variability that are divided into two categories: *hardware-related factors* [81] and *software-related factors* [128].

Hardware-related factors

Within a singular microarchitecture, execution time is subject to inherent variability arising from numerous intrinsic factors. In the following, we explore the potential factors contributing to this variability:

Memory hierarchy. Processors use various levels of cache memories to store frequently accessed data and instructions. When a code is executed multiple times, the cache may contain the required data/instruction, resulting in faster execution. However, if the needed data/instructions are not found in the cache (cache miss), the program will run slower (if there are no time anomalies [145]) because it has to rely on other types of memory, which are usually slower. The size and organization of the memories (cache and main memory), along with the code's memory access patterns, influence the cache hit and miss rates.

Pipeline dependencies. One source of the execution time variation is the pipeline, which refers to the sequence of stages or operations through which an instruction is divided, with each stage dependent on the completion of the previous one. The dependencies within the pipeline can be summarized into three types:

Read-after-write (RAW). One operation needs to read data produced by a previous operation before it can start.

Write-after-write (WAW). Multiple operations want to write in the same memory location, so they must be executed in a specific order.

Write-after-read (WAR). An operation needs to write data to a memory location that a subsequent operation wants to read from, so it must wait until the read is finished.

Branch prediction. Processors utilize branch prediction techniques to minimize the impact of conditional branches on program execution. If the processor's branch prediction mechanism accurately predicts the outcome of branches, it can maintain a high instruction throughput. However, if the predictions are incorrect, the processor may waste cycles fetching and executing instructions that are ultimately discarded, leading to longer execution times. The effectiveness of branch prediction depends on the specific patterns of conditional branches in the program.

Superscalar processors. A superscalar processor is a type of microprocessor that can execute multiple instructions in parallel. It achieves this by having multiple execution units, allowing it to process more than one instruction during a single clock cycle. Regarding pipeline dependencies, superscalar processors face similar issues as other pipelined architectures. The dependencies described above (RAW, WAW, and WAR) can cause stalls and inefficiencies in instruction executions, affecting the execution time. To mitigate these issues, superscalar processors use sophisticated techniques such as instruction reordering and dynamic instruction scheduling.

Out-of-order processors. Out-of-order processors are an advanced type of microprocessor that can execute instructions out of their original order as long as the data dependencies are maintained. In out-of-order processors, a large instruction window and a complex hardware structure are employed to detect and handle dependencies efficiently. The processor maintains a buffer called the reorder buffer to keep track of the order of the instructions in the original program sequence, ensuring that the instructions are committed to memory in the correct order (order of program instructions execution). When it comes to pipeline dependencies, this kind of processor can effectively reduce stalls and increase instruction throughput. By dynamically reordering instructions based on data availability, they can maximize the utilization of execution units, leading to improved performance and better overall efficiency.

CPU frequency scaling. Many modern processors employ dynamic frequency scaling [158] (DVFS), where the CPU clock speed can be adjusted according to the workload. If the processor detects a high demand for computational resources, it may increase its clock speed to provide better performance. On the contrary, if the workload is low, it may reduce the clock speed to save power. This variation in clock speed can affect the

execution time of a program, as a higher clock speed globally leads to faster execution.

Resource sharing. Improving system performance is critical, and sharing resources is the key to achieving this. Multicore processors, which allow multiple tasks to run simultaneously, are a significant advancement. This not only boosts system performance but also necessitates effective distribution of computing elements such as CPU cycles and memory bandwidth across running programs. However, this also presents challenges, especially when running resource-intensive tasks simultaneously. This scenario can lead to a shortage of available resources for each program, causing variations in execution times. Additionally, interference during memory hierarchy access and cache coherence issues [23, 153] can also lead to inconsistent execution times.

Other factors. Other factors can also influence execution time, such as **speculative fetching** [47], **variable latency instructions** [52] (e.g., square root, division), **multithreading** [14], and **timing anomalies** [145]. Additionally, the initial hardware state can set off a domino effect [14], further affecting performance.

Software-related factors

Factors not related to the processor microarchitecture can influence the execution time of programs. These software-related factors are:

Program inputs. Modifying the input of a program can affect its execution time. Different inputs may lead to different control flows, data access patterns, and, therefore, utilization of resources. For example, larger input sizes may require more memory or lead to more loop iterations, resulting in longer execution times, even when the hardware and software environment remains the same.

Compiler optimizations. The choice of compiler and its optimization settings can impact the execution time of a program. A compiler can apply various optimizations and changes to the program, such as loop unrolling, instruction scheduling, and constant folding, to generate more efficient code. Different compiler versions or optimization levels can result in different performance characteristics, affecting the execution times of the program.

Operating system interferences. The way the operating system schedules processes and assigns CPU time to processes can impact the execution time of a program. The scheduling algorithm, priority levels, and interruptions can influence how much CPU time is allocated to a specific program. If a program has lower priority or is competing with other high-priority processes (such as I/O interruption), its execution time may be longer.

These factors are crucial in determining the execution time of a task in different

scenarios. In an average case, it is essential to understand the common hardware state (for example, since cache hits occur more frequently than cache misses [81], a situation where the data is available in the cache can be considered as the average hardware state) and the typical software state (considering the most commonly used program inputs). Conversely, for the worst-case scenarios, it is necessary to identify the most unfavorable conditions that the hardware and software can encounter while running a program.

1.1.4 Execution time estimation techniques

Techniques used for execution time estimation can be classified in general into four categories: static analysis, measurement-based, simulation-based, and data-driven techniques.

Static analysis techniques

Static analysis techniques rely mainly on assumptions about the code's behavior complemented by the expert's knowledge to time a program. This approach involves an *automatic* and *detailed inspection of a program's code* without actually executing it. The aim is to make time complexity estimates based on the number and types of operations, data size, and control flow. A simple example would be to count the number of instructions and match this to an execution time. More complex solutions use more advanced cost functions for estimating the execution time of instructions as those provided by: Low-Level Virtual Machine Microprocessor Code Analyzer (LLVM MCA) [12], Open Simple Analytic Compiler Architecture (OSACA) [109], Portable inference of port Mappings for out-of-order processors by EVolutionary Optimization (PMEvo) [146], and Intel Architecture Code Analyzer (IACA) [92]. These tools analyze the execution of input assembly code using a static model of the processor and provide various statistics, such as throughput and latency.

Measurement-based techniques

These techniques are centered on collecting data derived directly from an observation of the system.

Profiling. Profiling is the practice of capturing various metrics during the execution of a program to analyze its performance and behavior. These metrics range from CPU usage and memory consumption to the frequency of specific operations. The insights gathered not only help in optimizing the program itself but also serve as a foundation for predicting the execution time of similar applications. Tools like the Performance Application Programming Interface (PAPI) [135] and Perf [49] are widely used for this purpose,

offering a comprehensive suite of functionalities to monitor performance attributes. Alternatively, hardware-based approaches, such as FPGA synthesis [136], can be employed to gather precise performance data.

Hardware solutions (e.g., Joint Test Action Group JTAG). These solutions involve dedicated hardware, such as JTAG interfaces, to capture hardware-level timing information. They offer accurate estimates but often necessitate additional hardware support. An example is the J-Trace Pro trace solution from Segger [151], which is used to connect to the JTAG interface of the target processor, alongside Ozone [71], a crossplatform debugger and performance analyzer. Ozone generates execution traces with a format of one line per machine instruction, including other information such as the cycle counter.

Simulation-based techniques

Simulations construct a replica of the processor microarchitecture behavior and estimate the execution time by running the simulated model under various scenarios. These simulators, such as GEM5 [19], ARM cycle accurate [120], uops info Code Analyzer (uiCA) [3], and simplescalar [13], can integrate heuristic simplifications (e.g., assuming some parts of the processor microarchitecture functioning and incorporate real world data, such as memory latency).

Data-driven techniques

Machine learning [126] presents a new paradigm for estimating execution time by leveraging historical data. Unlike traditional methods that depend on static models or broad approximations, machine learning algorithms, particularly regression models, delve into past code execution time records to predict future ones. These models excel in discerning the intricate relationships between code attributes and their corresponding execution times. For a comprehensive examination of related examples, refer to Section 1.3. It is worth noting that this approach forms the central focus of this thesis.

Each technique offers its own advantages and limits. The appropriate choice depends on a number of factors, including the available resources and the required precision of the timing estimate. The summary of the advantages and limits of the four categories of execution time estimation techniques is presented in Table 1.1.

Technique	Advantages	Limits
Static analysis	No need to execute code. Portable across different hardware.	Inaccuracies due to simplification. Resource-intensive for complex cases.
Measurement-based	Uses real system data. Captures complex behaviors.	Can be invasive. Limited by measurement scope.
Simulation-based	Emulates before implementation. Controlled environment for scenarios.	Needs accurate models. Simulations are slower than real execution.
Data-driven	Handles complex behaviors. Balances static assumptions with real-world data.	Needs quality historical data. Complexity can obscure interpretation.

Table 1.1 – Comparative analysis of execution time estimation solutions.

1.2 WCET estimation techniques

Estimating the ACET of a program is generally easier than determining its WCET. The central limit theorem [107] simplifies the ACET estimation task by offering a sound statistical basis. For instance, monitoring the execution time of a basic task on a simple processor for 100 iterations can provide a reliable estimate of the average time needed for execution. The actual average execution time can be approximated using the empirical average obtained from these 100 observations (i.e., Equation 1.1).

$$\bar{t} = \frac{1}{100} \sum_{i=1}^{100} t_i \tag{1.1}$$

However, determining the WCET is inherently difficult, and its undecidability [99] is rooted in the halting problem, a foundational issue in the theory of computation. Alan Turing proved in 1936 that a general algorithm to determine whether a given program (with a given input) halts or continues to run indefinitely cannot exist. This is known as the halting problem, and it is undecidable. In the context of WCET, determining the exact execution time would require knowing the maximum time a program takes for any possible input. If we could determine this, we would also know if the program halts for every possible input. But since the halting problem is undecidable, determining the exact WCET is as well. Therefore, while we can estimate the WCET for many programs,

especially those with bounded loops and inputs, we cannot create a general method that determines the WCET for all possible programs. Thus, to ensure the estimation is a practical value, it is imperative that it remains both safe and precise.

Definition 1 ***Safety:*** *An upper limit of the WCET of a task is considered safe if it is greater than all possible execution times.*

Definition 2 ***Precision:*** *A WCET estimate for a task is considered precise if it is close to the actual WCET.*

When estimating the worst-case execution time, there are several techniques available to provide safe and accurate predictions. These techniques can be classified into three main categories: static techniques, end-to-end measurement techniques, and hybrid techniques [171].

1.2.1 Static techniques

Static techniques [171] for WCET estimation process is depicted in Figure 1.1. It involves transforming code into a Control Flow Graph (CFG). This graph represents the potential execution paths of a program. For instance, Listing 1.1 and Figure 1.2 show, respectively, a C source code and its corresponding CFG that is extracted from the compiled binary code. In Figure 1.2, nodes correspond to basic blocks, and edges correspond to possible control flow between them. For example, the basic block *then* contains the code for *s=s+t[i]*. Static techniques generally involve the following stages:

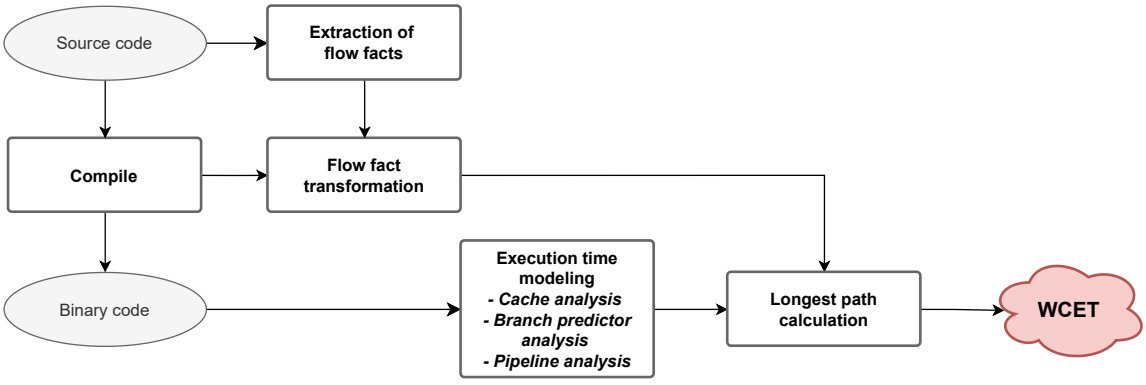

Figure 1.1 – Static timing analysis workflow.

Flow facts extraction and transformation. In this phase, various flow-related details are extracted from either the source code (for example, user-specified maximum loop iterations) or from the Control Flow Graph (CFG) extracted from the binary code.

Listing 1.1 – Example of C code

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

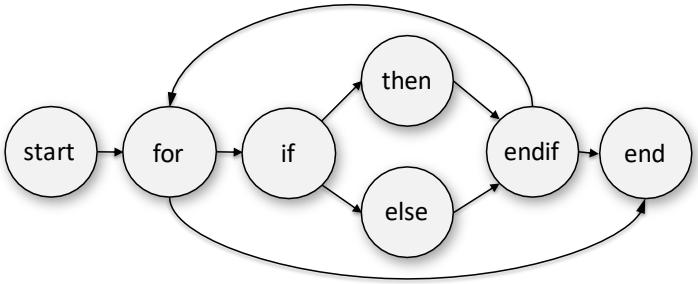

Figure 1.2 – Control flow graph for the code depicted in Listing 1.1.

Execution time modeling (HW abstraction). Also known as low-level analysis, this step involves statically determining temporal information about the worst execution time of each basic block. This is based on a model of the hardware architecture (which requires detailed documentation on the processor). The determination of the WCET of a basic block must take into consideration specific hardware elements such as pipelines [108, 113], cache memories [123, 68] and branch predictors [40]. These elements introduce variability in the execution time of instructions as discussed in Section 1.1.3, depending on the path taken within the program. Abstract interpretation [161] is usually used in this case to analyze the worst possible states without executing the program.

Longest path calculation (high-level analysis). This step employs algorithms designed to efficiently identify the path with the longest execution time without enumerating all possible paths. The identified path serves as an estimate for the worst-case execution time. The Implicit Path Enumeration Technique (IPET) [118] is commonly used at this stage. It formulates the longest path problem as an integer linear problem. Using the previous example (Figure 1.2 and Listing 1.1), we illustrate the WCET estimation calculation on the example that computes the sum of the absolute value of 100 elements stored in an array t using IPET.

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

Figure 1.3 – IPET formulas for the CFG of Figure 1.2.

— First, the WCET of a basic block b denoted by w_b is estimated by applying the HW abstraction model. 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 the basic block b.

— Constraints on variables n_b and $n_{\text{b}\to\text{b'}}$ (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 number 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 example of the constraints for the previous program is given in Figure 1.3. Assuming for the sake of illustration that the outcome of the learned 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 cycles.

For simplicity, we have assumed in this example that each basic block has a single, context-independent WCET estimate. For architectures with caches, pipelines, and branch predictors, this assumption is obviously no longer valid.

Despite the valuable contributions of static techniques in generating safe and deterministic worst-case estimates (if the proposed HW model is precise enough), they often produce conservative estimates, potentially leading to an overestimation of the execution time. These methodologies are implemented in both commercial tools such as AiT [66], and academic tools such as Heptane [78], Ottawa [15], Chronos [114], and SWEET [119].

1.2.2 End-to-end measurements techniques

End-to-end measurement techniques [53] for the estimation of WCET are based primarily on real measurements captured during the program's execution on the target hardware platform. The amassed measurements serve as the basis for estimating the WCET. Two notable measurement-based methodologies are:

Measurements with a safety margin. This approach requires running the program under its worst-case input and processor conditions to obtain the maximum execution time. The worst-case input refers to the input that results in the longest execution time,

while the unfavorable processor state might involve conditions like empty cache memories, flushed branch predictor, etc. Typically, a safety margin is added to the maximum observed execution time to account for uncertainties and unforeseen scenarios.

Statistical analysis. Particularly the application of Extreme Value Theory (EVT) [76], which plays a vital role in the estimation of measurement-based WCET. EVT specializes in analyzing timing measurements by concentrating on outliers—extreme deviations from the median in probability distributions. In the context of WCET, EVT is particularly adept at modeling the distribution of maximal observed execution times, thereby revealing crucial information about how the system—be it a program or a processor—behaves under extreme operational scenarios. This analytical approach is known as probabilistic WCET (pWCET) estimates [31, 28]. Unlike traditional WCET estimates that give a single deterministic value, pWCET provides a probability distribution.

1.2.3 Hybrid techniques

Hybrid techniques (e.g., [101]) integrate aspects of both static and measurement-based methods to estimate the worst-case execution time, as illustrated in Figure 1.4. In this approach, components of the static analysis process, such as flow facts extraction, transformation, and longest path calculation, are retained. However, the hardware abstraction phase is replaced with a measurement phase.

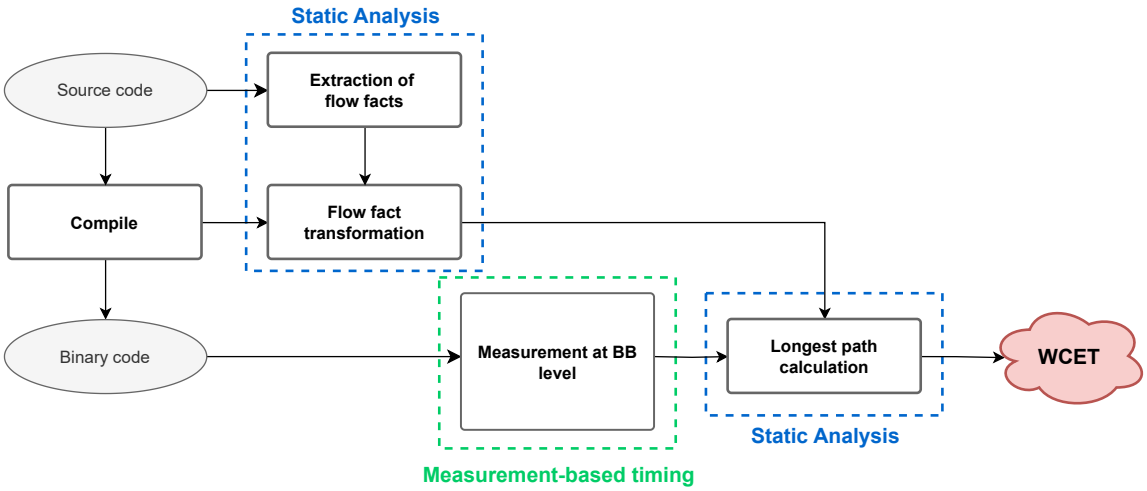

Figure 1.4 – Hybrid timing analysis workflow.

Examples of hybrid solutions for WCET estimation are Timeweaver [96] by AbsInt and Rapitime [48] by Rapita. These solutions employ a combination of hardware-assisted measurements, such as JTAG (Joint Test Action Group), and manual annotations, including waypoints, tracepoints, and interest points. These measurements are performed

on code snippets to determine their WCET. The static tool provided by these solutions then utilizes this information to estimate the WCET of the entire program.

Table 1.2 gives a high-level comparison of the three categories of techniques. Here, we show the advantages and limits of each category. Aside from accuracy and safety, the realtime systems community has increasingly emphasized two additional factors: performance requirements and architectural complexity. The adoption of sophisticated hardware architectures, often poorly documented due to intellectual property concerns, adds further challenges to each of the primary estimation techniques.

Technique	Advantages	Limits
Static	Safe and sound for hard realtime systems. Provides an upper bound for the WCET without executing the program.	Can be overly pessimistic, leading to overestimations. Hardware modeling needed for each new processor. Requires detailed documentation that might be unavailable.
End-to-end-based	Provides accurate (less pessimistic) estimates based on actual program executions. Captures the system's dynamic behavior.	Might not cover the absolute worst-case scenario, making them unsuitable for hard realtime systems. Measurements should be repeated for each new program.
Hybrid	Aims to balance the safety of static methods with the accuracy of end-to-end-based methods. Can provide more realistic WCET estimates.	Their effectiveness depends on a balance of static analysis and measurement data. Measurements at the BB level in hybrid methods can raise code coverage issues [111].

Table 1.2 – Comparing static, measurement-based, and hybrid WCET solutions.

Static techniques. Determining the WCET using static techniques, while being the most secure approach, comes with its own set of challenges. These methods require detailed hardware models, which in turn depend on comprehensive processor documentation. However, even when such documentation is available, the increasing complexity of modern hardware can lead to a phenomenon known as "state explosion" when using techniques like abstract interpretation. This state explosion refers to the rapid growth of possible states

that the system can be in, making the analysis computationally infeasible. As a result, applying static timing techniques becomes increasingly complicated on high-performance processors.

End-to-end-based techniques. Even if we assume that we can identify the program input that leads to the worst-case execution path, the hardware complexity makes it very difficult to initialize the processor to an accurate worst-case state, especially for the entire program.

Hybrid techniques. Hybrid methods afford the advantage of snippet-level measurements, capturing processor complexities more effectively. Coupled with flow analysis, these methods make the outcome independent of specific program inputs while offering some guarantees regarding prediction accuracy. This positions hybrid techniques as a relevant choice for complex processors with limited documentation if the problem of code coverage is solved. However, expecting users to procure both the processor and measurement hardware is far from ideal. This becomes even more evident when considering that the chosen processor might not have the resources or the required capabilities to support the realtime application. In this context, machine learning emerges as a promising solution. Not only can it mitigate the need for users to invest in expensive processors and measurement tools, but it also addresses the challenges of measurement code coverage [111]. Delving into a hybrid WCET estimation using a machine learning approach will be a primary focus of this document.

1.3 Machine learning

Machine learning [126] is a subdomain of artificial intelligence that has been widely adopted in many fields as an alternative approach to solve different problems. The relevance of machine learning arises from its strong ability to learn relationships between data, operating on what we call a *model* that learns from real world examples instead of relying on hard-coded rules.

Machine learning algorithms depend on various criteria, such as the degree of supervision provided (a.k.a supervised or unsupervised learning) or the way data are provided to the algorithm (e.g., reinforcement learning). In this thesis, we will focus on supervised learning.

Supervised learning. In supervised learning, the training process utilizes both input features and corresponding output targets to generate a model capable of predicting outputs for new, unseen inputs. Depending on the type of output, these models can be termed as *regression* for continuous numerical outputs or *classification* for discrete or categorical ones.

In supervised learning, our main interest lies in learning a model that links data $X = \{x_0, x_1, ..., x_n\}$ (where x_i is a feature of X) to a continuous label y, In the context of our study, this involves mapping the characteristics of a basic block to its execution time. Such a learning approach is termed regression. We define a regression model $F_{parameters}(X)$ as follows:

$$y = F_{parameters}(X) + E(X) \tag{1.2}$$

Here, *parameters* denote the parameters of the regression model, acquired during the training phase on $dataset_{train}$ $((X_1, y_1), ..., (X_k, y_k))$. Additionally, E is typically conceptualized as the discrepancy between the true value y and the output of the model $F_{parameters}$.

1.3.1 Regression-based machine learning algorithms

The process of training a regression machine learning model unfolds through several sequential stages, which are illustrated in Figure 1.5. It begins with data collection and cleaning. The cleaned data is then segregated into three distinct subsets: a training set $dataset_{train}$ (1a), a validation set $dataset_{valid}$ (1b), and a testing set $dataset_{test}$ (1c). The training set is integral for the calibration of the model's weights (Step 2a), primarily under the guidance of a loss function. Conversely, the validation and test sets are reserved solely for the evaluation of the model's predictive accuracy and robustness (Step 2b for validation and Step 5 for testing on unseen data). Following the initial training phase (Step 3a), the model's performance is assessed using the validation set (Step 3b). Should the model underperform, adjustments are made to the hyperparameters (Step 3c), and the model undergoes retraining and subsequent reevaluation. This cycle continues until satisfactory performance metrics are obtained, indicative of the model's accuracy and efficacy with the chosen hyperparameters [1]. Once satisfactory performance metrics are achieved (Step 4), the optimized model is ready for deployment on new, unseen data or the predefined testing dataset (Step 5). This step serves to identify any instances of overfitting that may have happened during the training sessions on the training dataset, ensuring the model's generalizability and reliability in real world scenarios.

To tune the weights of a regression model, several *loss functions* exist. In this thesis, we have considered the following ones (as a reminder, y_i is the ground truth value and

1. Hyperparameters are parameters that are not directly learned from the data. Instead, they are set prior to the training process and influence the behavior and performance of the model. For example: the number of neurons in a neural network.

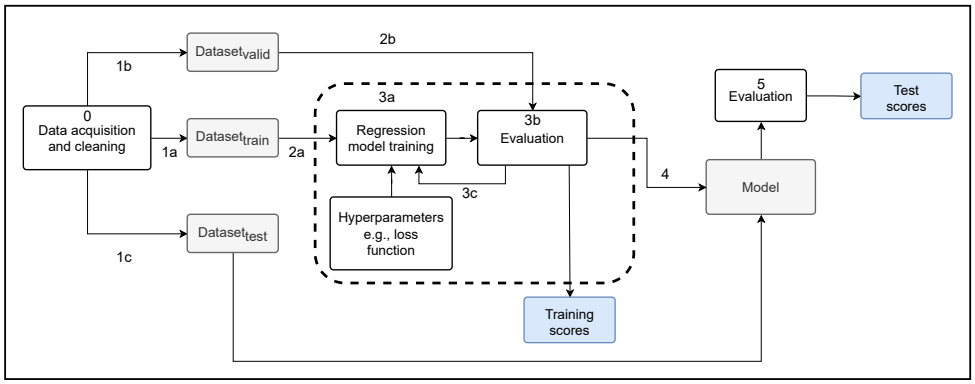

Figure 1.5 – Regression learning workflow.

$F_{parameters}(X_i)$ is the prediction):

— Mean Squared Error (MSE), defined as:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - F_{parameters}(X_i))^2 \tag{1.3}$$

— The Mean Absolute Percentage Error (MAPE), defined as:

$$\text{MAPE} = \frac{100}{N}\sum_{i=1}^{N}\left|\frac{y_i - F_{parameters}(X_i)}{y_i}\right| \tag{1.4}$$

— The Symmetric Mean Absolute Percentage Error (sMAPE), defined as:

$$\text{SMAPE} = \frac{100}{N}\sum_{i=1}^{N}\frac{|y_i - F_{parameters}(X_i)|}{(|y_i| + |F_{parameters}(X_i)|)/2} \tag{1.5}$$

— The Root Mean Squared Logarithmic Error (RMSLE), defined as:

$$\text{RMSLE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\log(y_i + 1) - \log(F_{parameters}(X_i) + 1))^2} \tag{1.6}$$

The MAPE function is employed to assess the model's accuracy on the validation (training scores in Figure 1.5) and testing datasets (testing scores in Figure 1.5). Meanwhile, the Pearson correlation coefficient r is used to evaluate the linear relationship between the predictions and the ground truth. The Pearson correlation coefficient is given by:

$$r = \frac{\sum(F_{parameters}(X_i) - \bar{F}) * (y_i - \bar{y})}{\sqrt{\sum(F_{parameters}(X_i) - \bar{F})^2 * \sum(y_i - \bar{y})^2}} \tag{1.7}$$

Where $\bar{F}$ represents the mean of all predictions.

In the following, we only present the regression models that will be used in the chapters detailing our contributions.

Linear Regression (LR) [69]. Linear regression is the most basic regression algorithm. It assumes a linear relationship between the input variables X and an output variable y. Ordinary Least Squares (OLS) is a type of linear regression that aims to minimize the sum of the squared residuals, i.e., the differences between the observed and predicted values. The resulting model is defined by the equation $y = a.X + b$, where a is a vector of weights for each element of X and b represents the bias. The values of a and b are estimated by the algorithm. An illustration of linear regression application in estimating execution time is presented in Figure 1.6. In this example, y symbolizes the execution time corresponding to a basic block, with x_1 denoting the number of instructions in the basic block, and x_2 representing the associated memory usage. From a graphical perspective, this linear regression model is represented by a plane that best fits the data points engaged during the parameter training phase. The resulting model serves as an analytical tool to analyze the linear relationships between the input variables and the output. Additionally, it can be directly employed to deduce the execution time based on the number of instructions and memory accesses.

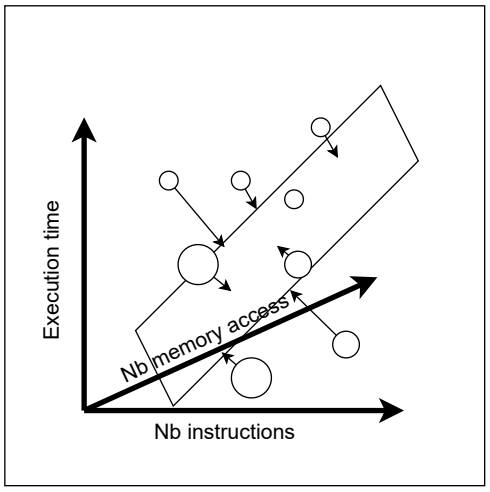

Figure 1.6 – Linear regression for execution time estimation example.

Capturing the relationship between the input variables X and the corresponding output y can be challenging when using a linear model, especially when it is aimed at minimizing errors solely on the training set. This approach may result in poor generalization

to unseen test data. To address these limitations, various extensions to traditional linear regression have been developed, including:

Polynomial Regression. An extension of linear regression that includes the powers of the input variables to model nonlinear relationships within a linear framework. For example, $y = \beta_0 + \beta_1.X + \beta_2.X^2 + \ldots + \beta_d.X^d + E(X)$

Ridge Regression [84]. Ridge Regression is an extension of linear regression that incorporates a regularization term. Its primary objective is to create a model that generalizes well to new, unseen data by constraining the model's complexity. It does this by adding a penalty based on the size of its parameters. The main idea is to not let any parameter become too dominant. The strength of this penalty is controlled by a value called λ. The modified loss function for Ridge Regression is defined as $loss_{adjusted} = Loss\left[(F_{parameters}(x_i) - y_i)\right] + \lambda \sum_{j=1}^{n} (weights_j)^2$. Where n is the number of features, and λ is the regularization parameter.

Lasso Regression [162]. Lasso Regression is another variation of linear regression. What is unique about Lasso is that it can completely remove some features (or parameters) if they are not that helpful. This makes the model simpler and easier to understand. It uses a different kind of penalty than Ridge, focusing on the absolute values of the parameters. Like Ridge, the strength of the penalty is controlled by a value called λ. The modified loss function for Lasso Regression is given by $loss_{adjusted} = Loss\left[(F_{parameters}(x_i) - y_i)\right] + \lambda \sum_{j=1}^{n} |weights_j|$. Where n is the number of features, and λ is the regularization parameter.

K-Nearest Neighbors (KNN) Regressor [102]. The K-Nearest Neighbors (KNN) regression algorithm estimates the value of a target variable by taking the average of the values of its K nearest neighbors. These neighbors are identified based on a distance metric, such as Euclidean distance, which is specified as a hyperparameter. Figure 1.7 illustrates this concept. In the example, we aim to predict the execution time of an unidentified point, denoted by a red dot. Using a specific distance metric, we identify 5 nearest neighbors (or 5 basic blocks) with known execution times. The predicted execution time for the red dot can be calculated in two ways. The execution time is either the average:

$$F_{(distance)} = \frac{1}{k} \sum_{i=1}^{k} y_i$$

or, each output y_i can be weighted by the distance d_i:

$$F_{(distance)} = \frac{\sum_{i=1}^{k} d_i y_i}{\sum_{i=1}^{k} d_i}$$

In both formulas, y_i represents the execution time of the i-th nearest neighbor, k is the number of neighbors, and d_i is the distance from the unidentified point to the i-th

nearest neighbor. The latter formula assigns more weight to neighbors that are closer to the point in question, thereby potentially improving the prediction accuracy.

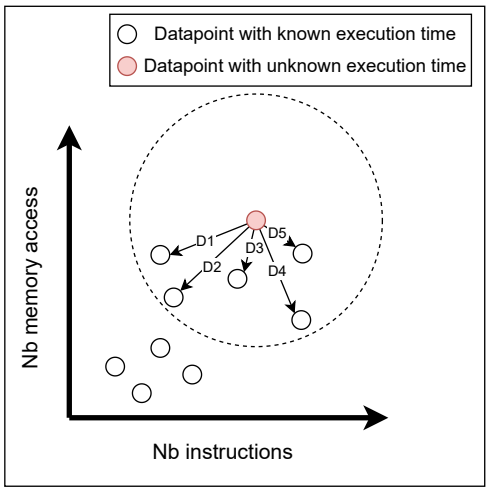

Figure 1.7 – KNN example for execution time estimation.

Decision trees [141]. They can be understood through their hierarchical structures, which arrange data in a tree-like form. In a decision tree, every internal node tests a specific feature, while the branches extending from it indicate the various possible values for that feature. The leaves of the tree serve as the predictions. When the model is used for prediction, it assesses new data by following the tree's branches according to the feature values of the input. Several techniques extend the concept of the decision tree to create more robust and higher-performing models, including:

Random Forest Regressor (RF) [42]. Random Forest is an ensemble method that combines a multitude of decision trees to make a prediction. In the context of regression, the final prediction is the average of the predictions of all the individual trees. It handles nonlinear relationships well and is robust to outliers. Figure 1.8 shows an example of multiple decision trees, where each tree gives a prediction of the execution time that will be averaged in the end with the predictions of the other trees.

eXtreme Gradient Boosting (XGBoost) [34]. XGBoost is an ensemble learning algorithm, similar in concept to Random Forest, but with a key difference in its approach to model construction. While Random Forest builds decision trees independently, XGBoost employs a sequential method that combines multiple trees to create a more accurate and robust predictive model. This can be metaphorically described as a collaborative effort of a "team of experts", where each subsequent decision tree focuses on correcting errors made by the preceding trees in the sequence. This iterative refinement makes XGBoost highly effective for a wide range of machine learning tasks.

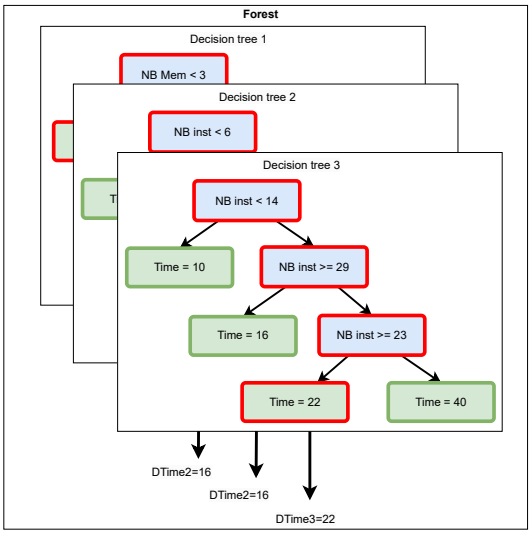

Figure 1.8 – Example of a random forest model for execution time estimation.

Support Vector Regression (SVR) [39] While the Support Vector Machine (SVM) algorithm is traditionally known for classification tasks, it can be adapted for regression through a variant known as Support Vector Regression (SVR). Similar to its classification counterpart, SVR aims to identify a hyperplane—or multiple hyperplanes in higher-dimensional spaces—that best represents the underlying relationship between the input variables and a continuous output variable, as illustrated in Figure 1.9. In contrast to SVM, where the goal is to maximize the margin between distinct classes, SVR seeks to closely fit the data points within a defined tolerance or "epsilon margin" e. Specifically, the algorithm aims to find a function $f(X)$ such that the deviation from each actual target value y_i in the training data is no greater than e. This enables SVR to produce a model that is both accurate and tolerant to small fluctuations in the data.

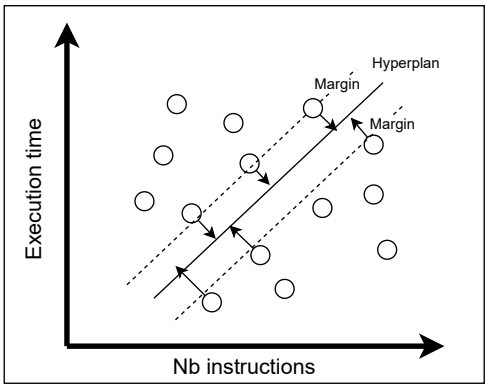

Figure 1.9 – SVR example for execution time estimation.

Deep Learning Models [80]. These models consist of interconnected neurons, with each neuron holding a function that combines inputs to produce an output. Their flexibility allows them to solve a wide range of problems using various architectures, which determine the layout of neuron connections in the network. Details about how these models are provided in the next Section.

1.3.2 Deep learning techniques

Deep neural networks (DNNs), also known simply as artificial neural networks, are graphs of computational units called artificial neurons. Neurons receive weighted input signals. They produce an output signal using an activation function.

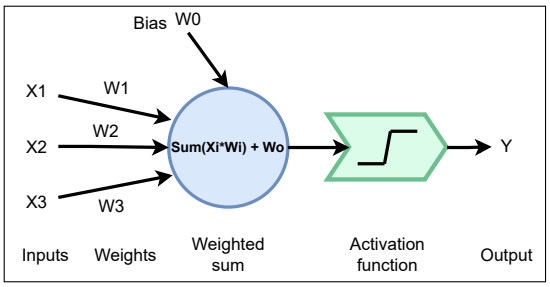

Figure 1.10 – An example of an artificial neuron.

Artificial neuron. Figure 1.10 illustrates an artificial neuron, with three input variables denoted as $X = \{x_1, x_2, x_3\}$. These inputs are each associated with a corresponding link weight $W = \{w_1, w_2, w_3\}$. Additionally, the model incorporates a bias term represented by the weight w_0. The output y of this neuron is computed as the weighted sum of its inputs, mathematically expressed as $y = (\sum_{i=1}^{n} w_i x_i) + w_0$. To make predictions, this output y is subsequently evaluated against a predefined threshold value. Specifically, y is transformed into an activation signal using either a threshold function or a nonlinear *activation function*. This activation signal serves as the input to subsequent layers of neurons in a neural network, if applicable.

Multi-Layer Perceptron. A common graph architecture consists of layers of neurons that form a complete bipartite graph between two consecutive layers: this is called a Multi-Layer Perceptron [2] (MLP), which is composed of three main parts (see Figure 1.11): an input layer, hidden layers, and an output layer.

Input layer. The input layer is the entry point for data into the neural network. It directly receives the raw data or features. Each node in this layer corresponds to one

2. A perceptron is the simplest form of a neural network, consisting of a single neuron or layer.

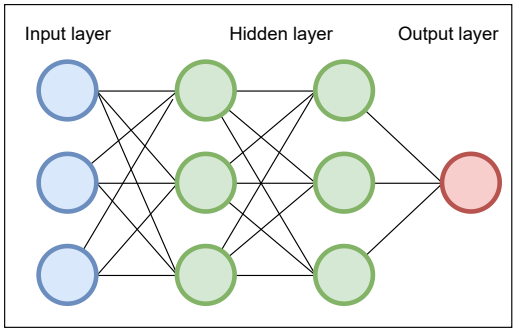

Figure 1.11 – An example of a deep neural network consisting of an input layer, hidden layers, and an output layer.

feature (or attribute) of the data. Essentially, it represents the initial data that you want to process or make predictions on.

Hidden layers. The hidden layers are where the magic of neural networks happens. These are called "hidden" because they are not directly exposed to inputs or outputs. These layers transform the data from the input layer through a series of weighted connections and activation functions. As the data moves through the hidden layers, the network learns and captures intricate patterns and relationships within the data.

Output layer. The output layer provides the final prediction from the network. It translates the complex processing done in the hidden layers into understandable predictions.

Forward propagation. When data is provided to the network, the outputs of all neurons in the first layer can be calculated by applying the previous formula (weighted sum plus bias) followed by the activation function. With the output from the first layer, we can calculate the output of the second layer, and so on, until we reach the final output. In this way, information is propagated throughout the network from the inputs to the outputs. This process is called "forward propagation". This is the same procedure used to make a prediction for a new input after the network has been trained.

Backpropagation of the gradient. During training, the output predicted by the model is compared with the expected output, and the resulting error is calculated using the *loss function*. This error is then propagated backward layer-by-layer, and the weights (corresponding to each graph edge) are updated based on their contributions to the error. We call this process "backpropagation". The process is repeated for all the examples (samples) or on a set of examples called *batch* in the training dataset. One pass through the entire dataset to train the neural network is referred to as one *epoch*, the dataset can be seen several times as the neural network can be trained for tens, hundreds, or even thousands of *epochs* to improve the model's training accuracy sequentially.

In the following, we present some important architectures of deep learning models that will be useful to understand this document and are better suited for temporal data types such as program codes.

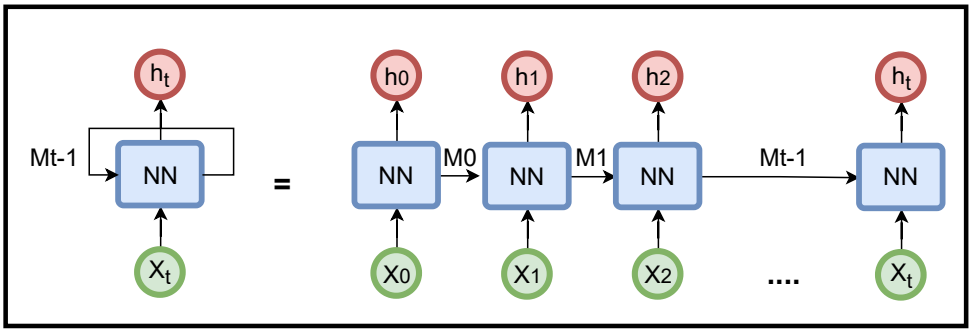

Figure 1.12 – An RNN unfolds through time.

Recurrent Neural Networks (RNN) [127]. RNNs are a specialized type of deep neural network tailored for sequence-based problems where the context or chronological order matters. Unlike Multi-Layer Perceptron, which allows data to flow in only one direction—from the input layer, through the hidden layers, to the output layer—RNNs facilitate the cyclic flow of information. This means an output from one step can influence the input of the next, making them capable of tasks where the sequence or context of input data is critical, as shown in Figure 1.12.

The left illustration of Figure 1.12 represents a basic RNN, while the right one shows the same RNN unfolded over time. For illustration, consider a basic block in assembly language comprising a sequence of instructions with inherent interdependencies. For instance, within a BB, we have the following instructions [LOAD R1], [ADD R1, R2], and [STORE R2]. At t=0, the "LOAD" instruction is processed as an input, depicted by X_0. The network processes X_0 to produce two results: h_0, the outcome from processing X_0, and M_0, indicating the memory or state after this instruction's processing. Next, at t=1, the register "R1" (from the LOAD instruction) is recognized as X_1. The network processes X_1 and integrates it with M_0 to generate outputs h_1 and M_1. By t=3, "R1" becomes the input, which, when combined with the state M_2, can help the RNN detect the dependency on "R1". The procedure follows, with each step's memory capturing the results of preceding processing stages, efficiently tracing the instruction dependencies within the network. Each state from a previous timestep, illustrated as $M_{(t-1)}$, is conserved and combined with the input of the forthcoming timestep, X_t. Such RNN configurations can be invaluable for analyzing and predicting the execution time of basic blocks.

Long Short-Term Memory (LSTM) Networks [82]. LSTM are an enhanced version of RNNs designed to address the long-term memory limitations of traditional

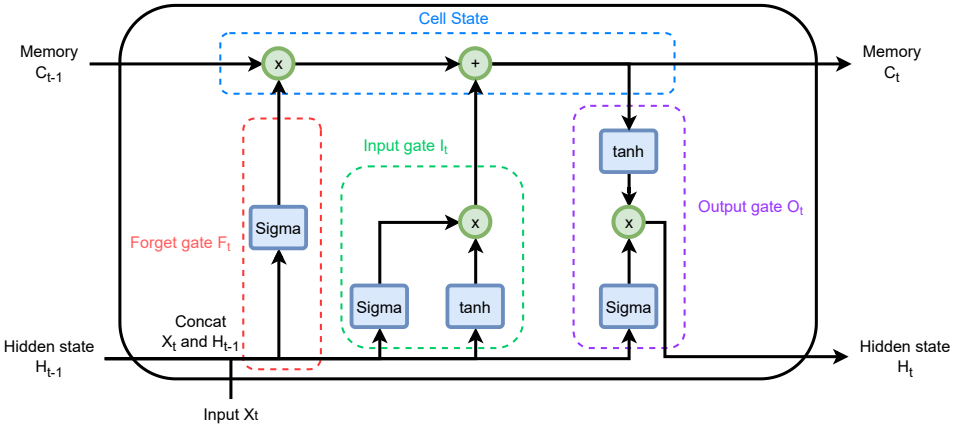

Figure 1.13 – An LSTM cell, represented by the different gates that compose it: Forget gate, Input gate, and Output gate.

RNNs (as RNNs tend to forget long-term dependencies due to the vanishing and exploding gradient problem [82]). An LSTM cell (Figure 1.13) features two types of memory: short-term memory H_t and long-term memory C_t. LSTM are mainly composed of three neural network blocks, each with a specific role, termed as gate. Each gate is a neural network that has a specific objective:

1. **Forget gate:** The first gate which determines which observations should be forgotten or removed from long-term memory;

2. **Input gate:** Which decides how much new information should be added to the memory cell C_t —also representing the actual output of the model at each step—;

3. **Output gate:** Which updates the short-term memory H_t.

To better understand the LSTM, we give the following analogy about a library. Imagine an LSTM as a library where books are being processed. C_t represents the actual collection, H_t represents book recommendations, which depend on the demands of library-goers in the previous days, and X_t represents the demands of the library-goers for the actual day. Each gate will be represented by a librarian as follows:

— Forget gate (Librarian for old books): This librarian manages the collection of old books (previous information). He decides which books are outdated or no longer needed and removes them from the shelves, and keeps the books that are still relevant, using the actual demands information X_t and historical demands list during previous days H_t.

— Input gate (Librarian for new arrivals): This librarian is in charge of new book arrivals (incoming information). He assesses the value of each new book and decides which ones to add to the collection, depending on their relevance and the existing collection. Finally, he updates the shelves with new valuable books.

— Output gate (Librarian for lending): Based on the available collection (current state

of memory cell C_t) and the demands of the library-goers (new inputs X_t), this librarian decides which books to recommend H_t. This information is used the next day by other librarians to make decisions.

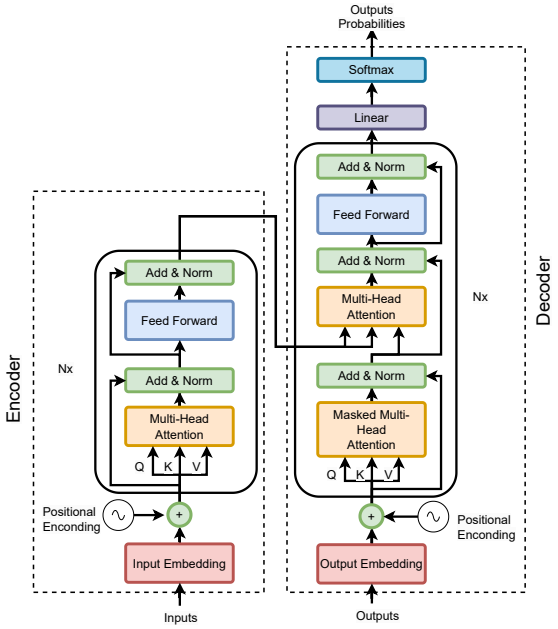

Figure 1.14 – The Transformers architecture, as described in the original paper.

Transformers [164]. The Transformers, introduced by Vaswani et al. in 2017, has marked a significant shift in the paradigm of sequential data treatment, outclassing the performances of traditional RNNs and LSTMs. One of the defining characteristics of Transformers is their ability to process the entire input sequence simultaneously. This global perspective allows Transformers to capture long-range sequences and all data dependencies more effectively. Instead of relying on recurrent connections like their predecessors, Transformers employ an *attention mechanism*. This mechanism can be visualized as a spotlight, emphasizing specific portions of the input sequence, enabling the model to concentrate on the most pertinent information.

Figure 1.14 depicts the Transformers architecture, which is primarily composed of two components: the encoder and the decoder. While the decoder is responsible for sequence generation, it falls outside the purview of this document. Our focus will be on the encoder. To simplify the explanation of how encoders work, we will explain two major components: positional encoding and self-attention mechanism.

Positional encoding. Unlike traditional sequential models that process inputs step-by-step, the encoder in the Transformers takes in the entire sequence simultaneously. However, this poses a challenge: without any inherent notion of sequence order (as there is no step-by-step processing), how does the model differentiate between the positions of

elements in the sequence? This is where positional encoding comes in. It provides the model with a unique signature for each position in the sequence, ensuring that the model can recognize the order of elements using this signature. For illustration, consider a simple assembly instruction: MOV R2, R3. Initially, each word is transformed into embeddings, yielding the following:

$$\text{MOV: } [0.1, 0.9, 0.3], \quad \text{R2: } [0.8, 0.4, 0.2], \quad \text{R3: } [0.7, 0.6, 0.5]$$

Subsequently, positional encodings—vectors formulated to be combined with word embeddings—are appended to these initial embeddings. For the sake of simplicity, we can consider the positional encodings as follows:

$$\text{Position 1: } [0.01, 0.01, 0.01], \quad \text{Position 2: } [0.02, 0.02, 0.02], \quad \text{Position 3: } [0.03, 0.03, 0.03]$$

After addition, the modified embeddings are:

$$\text{MOV: } [0.11, 0.91, 0.31], \quad \text{R2: } [0.82, 0.42, 0.22], \quad \text{R3: } [0.73, 0.63, 0.53]$$

These refined embeddings are then fed into the model, allowing it to discern that "MOV R2 R3" and "MOV R3 R2" are distinct instructions due to the differing register orderings (which is important for handling register dependencies problems), despite the identical word set.

Attention mechanism. The encoder contains multiple self-attention layers, as indicated by "N" in Figure 1.14. These layers allow the model to weigh the significance of distinct parts of the sequence differently, enabling it to focus on the most relevant parts at any given time. Consequently, as we move through the layers of the encoder, each one captures a progressively more abstract representation of the input, resulting in a comprehensive and multi-faceted understanding of the entire sequence.

Context fragmentation problem. Transformers have a shortcoming when it comes to handling exceptionally long sequences. Due to memory limitations, a Transformers model is constrained by a fixed maximum input length. Sequences exceeding this limit must be divided into smaller fragments that conform to the model's input size constraints. This fragmentation disrupts the model's understanding of the sequential context.

Transformers XL [43]. Transformers XL, as illustrated in Figure 1.15, marks a significant advancement in Transformers architectures by addressing the challenge of *context fragmentation*. One of its standout features is its capacity to "recall" or remember previously treated fragments of data. Instead of processing each fragment in isolation, Transformer XL integrates information from previous fragments, using this accumulated

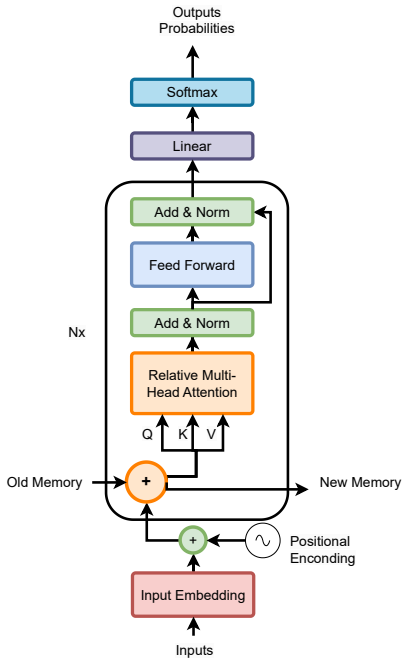

Figure 1.15 – The Transformers XL architecture.

knowledge as a foundation when interpreting new data. This continuity is facilitated by a technique known as the "recurrence mechanism", which is somewhat akin to the workings of RNNs or LSTMs. In Figure 1.15, the "Old Memory" represents retained information, while the "New Memory" captures the latest processed data. As data flows through Transformers XL, these two components interact, ensuring a holistic understanding of sequences and interconnecting even distant pieces of information.

Training effectively a Transformers. Training a Transformers efficiently usually involves two phases: pretraining and finetuning.

1. **Pretraining** [62]: The pretraining phase is a critical step in training Transformers models, especially in the context of language modeling. During this stage, the model is trained on a voluminous corpus of text data in a self-supervised way, enabling it to grasp the linguistic structure, semantics, and more intricate language patterns. One prevalent approach to pretraining is Masked Language Modeling (MLM) [57]. In this method, a specific proportion of the input tokens are randomly masked, and the model is trained to predict these masked tokens based on their surrounding context. For instance, Figure 1.16 illustrates this concept with the sentence "MOV ____ *Constant*, ____ *Adress*," where the model aims to predict the masked words "Register" and "Branch" based on the adjacent tokens[3]. This practice enables the model to develop robust language representations in the hidden layer of the neural

3. In Natural Language Processing, a "token" refers to an individual piece of text such as a word, a number, or punctuation.

network.

2. **Finetuning**: After the pretraining phase, the Transformers model acquires a foundational understanding of the domain-specific language and structure. However, to tailor the model for specialized tasks—such as execution time estimation—a finetuning phase is necessary. This process entails additional training on a smaller labeled dataset that is pertinent to the specific targeted task. During this finetuning phase, the model refines its generalized domain knowledge to suit the nuances and requirements of the targeted application.

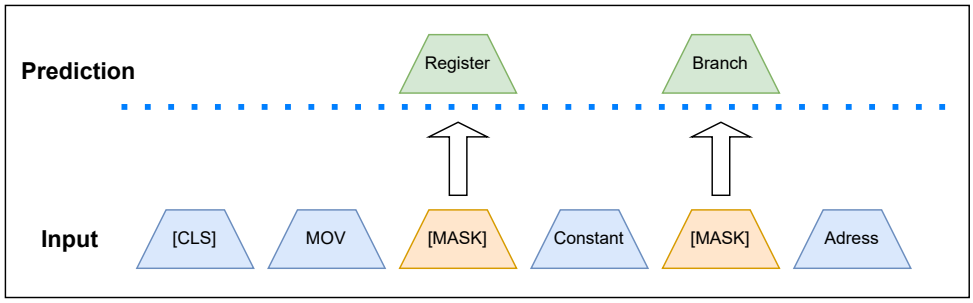

Figure 1.16 – Masked language modeling on a simple example.

1.3.3 Inputs used for ML-based timing models

The categorization of machine learning techniques usable for timing estimation can be based on the representation of the code to analyze (extracted static attributes, dynamic attributes, or embedding representations of code). In the following sections, we delve into each level and category.

Extracted static features a.k.a. "handcrafted features"

Some machine learning methodologies focus on feature extraction from code or system attributes believed to have an impact on execution time [138]. These extracted features can encompass architectural variables (e.g., cache size, memory bandwidth) or instruction-level specifics (e.g., number of arithmetic operations, memory access operations, or branching operations). Traditional methods of feature engineering [4] can be applied to select these attributes. Table 1.3 provides an illustrative example of static features that might be harvested from a given program. For instance, the occurrence frequency of each instruction within the code can serve as a distinguishing characteristic of the program.

4. Feature engineering is the process of selecting, transforming, or creating relevant input variables (features) to enhance the performance of machine learning models.

Feature	Value
ft1	Number of instructions
ft2	Number of add instructions
ft3	Number of sub instructions
ft4	Number of mult instructions
ft5	Number of div instructions
ft6	Number of load instructions
ft7	Number of store instructions
ft8	Number of comparisons
ft9	Number of conditional branches
ft10	Number of unconditional branches

Table 1.3 – Representing a code snippet with static features [138].

Extracted dynamic features

Performance counter-based techniques leverage hardware performance counters to collect fine grained information about the system's behavior during program execution [138]. These techniques capture events such as cache hits, branch mispredictions, or memory accesses. By incorporating performance counter data as input, machine learning models can learn from the underlying hardware behavior and improve estimation accuracy in some cases. Table 1.4 presents an example of the performance counters that PAPI [135] provides.

Counter	Description
PAPI_TOT_CYC	Total cycles
PAPI_TOT_INS	Total instructions
PAPI_BRI_TKN	Branch instructions taken
PAPI_BRI_NTK	Branch instructions not taken
PAPI_BR_MSP	Branch mispredictions
PAPI_LD_INS	Load instructions executed
PAPI_SR_INS	Store instructions executed
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_FP_INS	Floating-point instructions executed
PAPI_VEC_INS	Vector/SIMD instructions executed
PAPI_RES_STL	Cycles stalled on resource contention
PAPI_LD_INS	Load instructions executed
PAPI_SR_INS	Store instructions executed
PAPI_TLB_DM	Data Translation Lookaside Buffer (TLB) misses
PAPI_TLB_IM	Instruction TLB misses
PAPI_TOT_CACHES	Total cache accesses
PAPI_TOT_INS_I	Total instructions completed

Table 1.4 – List of some PAPI performance counters [135].

Embedding representations of code

An alternative strategy in machine learning focuses on directly learning code representations. Instead of manually extracting features, which can be tedious and error-prone, this method allows the machine learning model to automatically capture the essential features and characteristics of the code. Techniques like Word2Vec [133] or Bidirectional Encoder Representations from Transformers (BERT) [54] are employed to transform code snippets or entire program structures into continuous vector representations, often referred to as *embeddings*. These embeddings, once generated, serve as input for machine learning algorithms. They offer a comprehensive view of the code, capturing both its local nuances and broader structures. This holistic understanding allows the model to recognize intricate relationships and dependencies within the programming constructs, leading to more accurate and insightful predictions.

In the literature, we can find different techniques to embed codes:

One-hot encoding. One-hot encoding is a foundational method for converting categorical variables into binary vectors. In this approach, each distinct category within a variable is represented as an individual binary feature. Specifically, each category is converted into a binary vector whose length matches the total number of categories. In this vector, all elements are set to "0", except for a single "1" that marks the presence of the category in question. For instance, as illustrated in Figure 1.17, a vocabulary table is utilized to construct a lookup table, where each row represents an ARM assembly instruction as a binary vector.

id	Instruction
1	Add
2	Sub
3	Mul
4	Div
5	Mov
...	...
N	Store

N = Vocabulary size

id	Add	Sub	Mul	Div	Mov	...	Store
1	1	0	0	0	0	0...0	0
2	0	1	0	0	0	0...0	0
3	0	0	1	0	0	0...0	0
4	0	0	0	1	0	0...0	0
5	0	0	0	0	1	0...0	0
...	0	0	0	0	0	0...1	0
N	0	0	0	0	0	0...0	1

One hot encoding

Figure 1.17 – Basic block representation using one-hot-encoding example.

Word2Vec [133]. Developed by Google, Word2Vec is a method that transforms words into numerical vectors, essentially giving each word a unique numerical fingerprint based on its context and meaning. It utilizes neural networks and is grounded in the idea that words appearing frequently together in texts are likely to have related meanings.

Two primary strategies are employed in Word2Vec: Continuous Bag of Words (CBOW) and Skip-Gram. Both are visualized in Figure 1.18. To understand them better, consider the following sentence in natural language, "The cat sat on the mat".

CBOW. CBOW predicts the embedding of a target word based on the surrounding words. For instance, given the surrounding words "The", "cat", "on", "the", it tries to predict the embedding of the word "sat". This method, depicted on the left side of Figure 1.18, is particularly effective for capturing the general context around a word.

Skip-Gram. Skip-Gram operates in the opposite manner to CBOW. Given the target word, it predicts the embeddings of the surrounding words. For example, starting with the word "sat", Skip-Gram would predict the embeddings of surrounding words like "The", "cat", "on", "the". This method, shown on the right side of Figure 1.18, is adept at understanding the specific contexts in which a word can appear.

In short, while CBOW uses context to predict a word's embedding, Skip-Gram uses a word to predict the context embeddings. In this document, the focus will be on the usage of CBOW. The rationale for this choice, especially when dealing with assembly code, is that CBOW tends to be more efficient with larger datasets. Assembly code, being low-level and verbose, often results in extensive datasets.

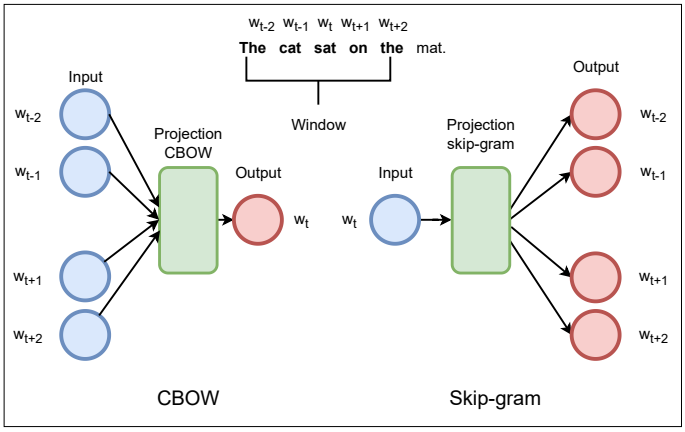

Figure 1.18 – Word2vec architecture (CBOW and Skip-gram).

Transformers. BERT [57] is a specific model based on the Transformers architecture. One of its standout features is its ability to capture the context and relationships between different parts of a sequence. This ability is especially valuable for generating good embeddings using attention mechanisms. Building on the capabilities of BERT, techniques like CodeBert [65] and PalmTree [115] have been successfully developed to create embeddings specifically for code. Figure 1.19 provides a visual representation of this attention mechanism in action for the input "MOV R2, #1, BR 0xF124". The resulting attention matrix showcases how different parts of the code influence each other, preserving the relational information between instructions. Such embeddings, rich in contextual information, prove invaluable for various tasks related to code, including generating new code, annotating existing code, or even correcting errors. In our case, we will use it to represent the basic blocks and estimate their execution time.

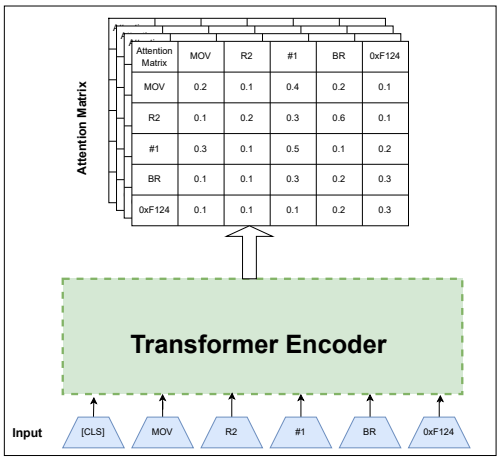

Figure 1.19 – Assembly code embedding using Transformers's attention matrix.

1.4 Machine learning for execution time estimation

Table 1.5 provides a summary of the works conducted for estimating execution time based on different scenarios: average, best, and worst cases. It includes information about the code level abstraction (source, intermediate, or binary), measurement tools used to capture the execution time, machine learning methods employed, types of inputs utilized, and dataset sizes. In general, we can make the following observations:

— Most studies depicted in the table concentrate on source code for analysis without any consideration of compiler effect. Linear regression (LR) and neural networks (NN) emerge as commonly employed ML algorithms and more sophisticated ones like LSTM and Transformers are rarely used.

— Reviewing the execution time measurement tools reveals a noteworthy inclination towards software (SW) measures, which is problematic due to the *probe effect* [5]. However, some studies incorporate hardware (HW) measures or simulations.

— The variation in dataset sizes is also important, with "small" dataset sizes being predominant, reflecting possible limitations in data acquisition in this field.

5. Probe effect refers to the phenomenon where the act of observing or measuring a system alters the behavior of that system.

Paper	Case	Code level abstraction	Measurement tool	ML algorithm	Inputs	Dataset size
[166]	Average	Source code	SW measures	LR, NN	Performance counters and compiler settings	Small
[46]	Average	intermediate representation	SW measures	LSTM	MLIR instruction and auxiliary HW input	Huge
[18]	Average	None	SW measures	LR, SVR, KNN	Input data features and performance counters	Small
[9]	Average	Source code	SW measures	LR, SVR, RF	Extracted features and performance counters	Small
[10]	Average	Source code	SW measures	LR, SVR, RF	Performance counters	Small
[150]	Average	Binary	HW measures	LR, RF, XGBoost, NN	Performance counters and static features from LLVM-MCA	Huge
[60]	Average	Source code	SW measures	LR, NN	Performance counters	Small
[138]	Average	Source code	SW measures	LR, SVM, NN	Instructions	Small
[24]	Average	Source code	SW measures	SVM, NN	Static features of CNNs	Medium
[25]	Average	Source code	SW measures	LR, NN, SVM, RF, XGBoost	Static features of CNNs	Medium
[159]	Best	Binary	SW measures	GNN	Instructions	Huge
[129]	Best	Binary	SW measures	LSTM	Instructions	Huge
[155]	Best	Binary	SW measures	Transformers	Instructions	Huge
[105]	Worst	Source	SW measures	LR, SVR, RF	Type of instructions	Small
[104]	Worst	Source	GEM5 simulator	LR, SVR, RF	Type of instructions	Medium
[87]	Worst	Source	HW/SW measures	LR, RF, SVR, KNN	Type of instructions	Small
[90]	Worst	Source	HW/SW measures	NN	Type of instructions	Small
[21]	Worst	Source	Static analysis	WEKA: LR, SVR, RF, NN	Type of instructions	Small

Paper	Case	Code level abstraction	Measurement tool	ML algorithm	Inputs	Dataset size
[73]	Worst	IR	SW measures and GEM5 simulator	LR	Type of instructions	Small
[131]	Worst	Source	SimpleScalar simulation	SVR	Type of instructions	Small
[132]	Worst	Source	Chronos and SimpleScalar simulation	SVR, NN	Type of instructions	Small
[154]	Worst	Source	GEM5 Simulation	LR, NN, SVR	?	Medium
[11]	Worst	Source	SW measures	NN	Function parameters	Large

Table 1.5 – Summary of works conducted for estimating the execution time.

In the following, we will discuss the most relevant works, categorizing them according to the execution time scenario they address (ACET, BCET, and WCET).

1.4.1 ACET estimation using ML

Most of the research on average-case scenarios focuses on improving compilation performance and uses basic machine learning techniques such as linear regression, support vector machine regressors, and/or random forest regressors. One notable work in this area is presented by Huang et al. [85]. They employ polynomial regression to predict program execution time based on static features such as loop counts, branch counts, and variable values. In addition to considering static features, some researchers have focused solely on using performance counters for execution time estimation. The work [10] proposes a methodology that solely relies on performance counters. By analyzing them, the authors aim to predict the execution time and evaluate the effectiveness of compiler optimizations. Another relevant study by Marcos Amaris et al. [9] explores the use of machine learning techniques for execution time prediction on GPUs. They utilize a combination of performance counters, such as the number of cache accesses and main memory accesses, along with static features of the application, including the number of basic blocks and threads. The authors employ LR, SVR, and RF algorithms to accurately predict execution time.

1.4.2 BCET estimation using ML

In the field of best-case execution time, research is primarily focused on estimating the execution throughput of x86 architectures. ITHEMAL [129] is one of the pioneering works that introduced RNN models for time estimation. ITHEMAL utilizes a hierarchical multiscale LSTM layer to accurately forecast the throughput of basic blocks. The RNN LSTM-based module of ITHEMAL captures intricate relationships among instructions within the same basic block. Figure 1.20 illustrates the architecture of the LSTM models used in ITHEMAL. Initially, the basic block is divided into instructions, and each instruction is further separated into operations or operands. These components are assigned a fixed number (a predefined token from a dictionary). This simplifies the process of embedding using word2vec [133]. The embedding of each instruction component is handled by the first LSTM layer. The second layer, on the other hand, focuses solely on the final representation of the entire instruction to create a representation of a basic block. The basic block representation is then passed to a feedforward layer to predict the throughput. BHive [37] is the dataset used to train ITHEMAL. This dataset consists of isolated basic blocks that are prepared under optimal conditions to measure processor performance (removing branching, ensuring that memory accesses are handled at the first level of cache,

etc.). The isolated basic block is then executed repeatedly until it reaches steady-state behavior. DeepPM [156], in the same fashion as ITHEMAL, predicts the execution time of a basic block in isolation using a simplified Transformers architecture. The lack of details on the paper and the unavailability of the code made the description of this work infeasible.

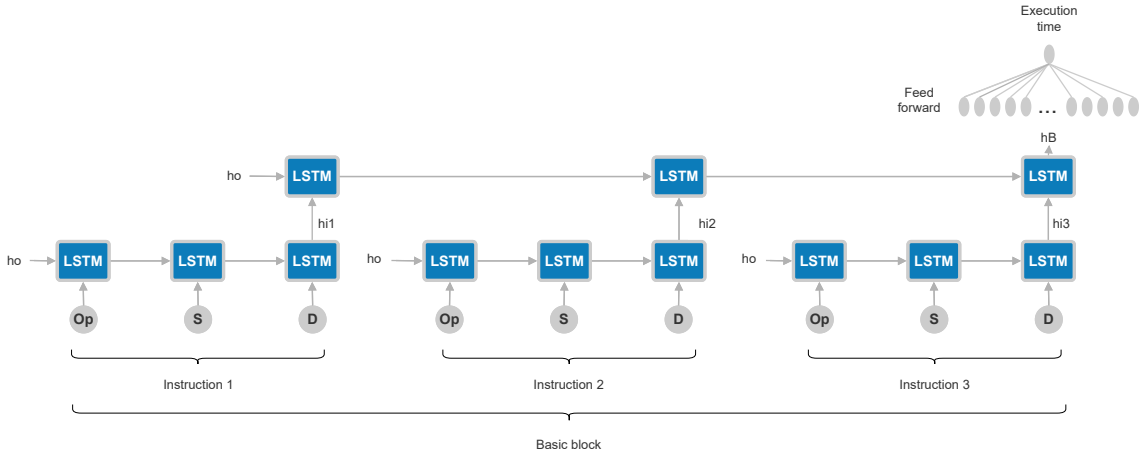

Figure 1.20 – ITHEMAL architecture.

1.4.3 WCET estimation using ML

In the worst-case scenario, two types of approaches can be distinguished. The first employs machine learning for end-to-end program measurement. Notable works in this category, such as those by Gustafsson et al. [73] and Bonenfant et al. [21], rely on an intermediate code representation to extract program attributes. These attributes are then employed in trained machine learning models to estimate the WCET. The second approach is presented by the work of Thomas Huybrechts [87, 90]. Huybrechts introduced a hybrid methodology. This strategy uses both static and machine learning-based methods, aiming for a reasonable blend of computational efficiency and predictive accuracy. In this framework, the source code is partitioned into segments called "hybrid blocks", characterized by single entry and exit points[6] and ranging from individual instructions to entire functions. Machine learning-based estimations are conducted for each of these blocks, and the outcomes are integrated statically to form a composite WCET estimate. This hybrid approach is incorporated into the Code Behavior fRAmework (COBRA) tool [89], an open-source platform designed for various resource optimization tasks, including WCET analysis, scheduler tuning, and multicores performance enhancement. Preliminary results

6. Unlike the definition of a basic block, which is a sequence of instructions without any branching in the middle, hyper blocks can have these branches within them. Thus, a hyperblock is a subgraph of basic blocks where one can enter from a single point and exit from a single point.

indicate that this hybrid methodology substantially mitigates the analytical overhead associated with static and measurement-based techniques while maintaining WCET predictions that closely approximate actual values. It is worth noting, however, that this method does not account for compiler effects, as it operates directly on source code features for basic block WCET prediction.

1.5 Conclusion

The background provided delved into different execution time estimation methods, with machine learning being a prevalent tool across average-case, best-case, and worst-case scenarios. Various algorithms, ranging from linear regression to advanced ones like LSTM, highlight the progressive nature of "machine learning for timing estimation".

WCET estimation stands out due to its critical role in real-time systems. Huybrechts' hybrid approach [88, 91, 87], which merges static analysis and machine learning, is noteworthy. While it seeks a balance between computational efficiency and prediction accuracy, its focus on source code features over compiler effects could be a drawback.

The summary of related studies in Table 1.5 showed that many works overlook hardware complexity and the interplay between instruction sequences and hardware (pipeline and cache effects). This oversight can introduce bias and inaccuracies when training machine learning models, especially when it is applied at the source or intermediate code stages (therefore ignoring the compiler optimization effects). Moreover, the precision of machine learning predictions can be compromised by the often limited size of training datasets. These observations prompt two unresolved questions: Can machine learning methods be tailored to work efficiently with modern processors, especially given the increasing prevalence of these processors? And how can machine learning models be trained to account for instruction dependencies to yield more accurate timing estimates?

In conclusion, leveraging machine learning for execution time estimation is a promising approach. However, careful consideration of the challenges, including hardware complexity, limited training data, and the need for precise modeling of interactions between instruction sets and hardware, is crucial for the successful application of machine learning in this domain. This document will therefore focus on addressing these challenges and exploring new solutions to enhance the accuracy of machine learning-based execution time code estimation.

Chapter 2

WCET Estimation Using Classical Machine Learning Techniques

In this chapter, we introduce WE-HML, a novel hybrid approach for Worst-Case Execution Time (WCET) estimation. WE-HML stands for **W**orst-Case **E**xecution Time Estimation using a **H**ybrid **M**achine **L**earning-based technique). This marks the first attempt in this thesis to combine machine learning with WCET estimation, specifically through the use of traditional machine learning methods such as linear regression, random forest, or neural networks. It serves as a foundational work, laying the foundations for the subsequent research in this document.

The WE-HML method uses machine learning in its two main phases: learning and WCET estimation. In the *learning phase*, machine learning models are trained using timing data from different basic blocks to predict WCET. These predictions account for varying execution contexts. During the WCET *estimation phase*, the trained models calculate the WCET for each basic block of a program, considering cache effects. The program's overall WCET is then determined using an adapted IPET approach.

The main contribution of this research is the development of a novel hybrid WCET estimation technique tailored for single-core processors. This technique relies on a machine learning-derived timing model for the processor's core and accounts for processor cache behavior, requiring minimal information about the processor's memory hierarchy. This work was accepted and published at RTCSA 2021:

"Abderaouf N., AMALOU, Isabelle Puaut, and Gilles Muller. "WE-HML: Hybrid WCET Estimation Using Machine Learning for Architectures with Caches." The 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). IEEE, 2021."

Hypotheses

In this subsection, we outline the hypotheses and operating assumptions that form the foundation of the WE-HML technique.

— WE-HML is particularly well-suited for applications requiring intermediate levels of safety assurance, such as those falling within the DAL B and C categories in the aeronautics industry [20]. These applications often run on complex processors for which creating a reliable timing model is challenging, rendering traditional static WCET estimation methods impractical. While some level of pessimism in WCET estimates is tolerated for these systems, missing a deadline is considered acceptable if it occurs rarely.

— In terms of methodology, WE-HML learns the WCET for each basic block under varying *execution contexts*. Specifically, these contexts take into account the level of *data cache pollution* generated by the concurrent execution of other code within the same loop nest. A unique advantage of WE-HML lies in its minimal reliance on detailed knowledge of the memory hierarchy, simplifying its implementation and application.

— In our approach, a sequence of instructions executed consecutively is broadly categorized as a basic block. This categorization allows us to incorporate the effects of branching into the WCET estimates for the generated basic blocks.

The remainder of this chapter is structured as follows. Section 2.1 provides a comprehensive introduction to the WE-HML technique. Section 2.2 outlines the experimental methodology employed for evaluating WE-HML on an ARM Cortex-A53 processor. Section 2.3 then presents the results obtained from these experiments. Finally, Section 2.4 provides a critical discussion of the findings and explores the limitations of WE-HML.

2.1 The WE-HML approach

WE-HML operates in two phases. The first phase, where machine learning algorithms are trained using measurements on a variety of basic blocks, is described in Section 2.1.1. The second phase estimates the WCET of programs using a modified IPET calculation method and is discussed in Section 2.1.2. Section 2.1.3 details the automatic generation of training data, and the support of caches is elaborated in Section 2.1.4.

2.1.1 Learning the processor timing model (training)

The training phase is executed once for each target architecture with the primary objective of learning the processor timing model, as depicted in Figure 2.1. WE-HML incorporates five machine learning algorithms, and our evaluations utilized algorithms from the Scikit-learn library [139, 70]. Initial experiments guided our focus toward the

top five algorithms that demonstrated the most promising outcomes. These algorithms are listed in Table 2.1, with comprehensive details provided in Chapter 1, page 38.

These algorithms are trained on basic blocks that are automatically generated, as detailed in Section 2.1.3. The automatic generation of basic blocks aims to cover a large variety of code structures in real codes. 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.

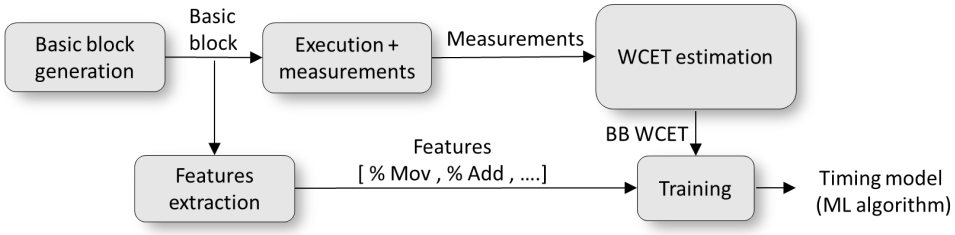

Figure 2.1 – WE-HML training phase.

The ML algorithms learn from the values of numerical quantities, called *features*. For the scope of this chapter, the features used are handcrafted. 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 allows the construction of a timing model 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.

Table 2.1 – Experimented machine learning algorithms.

Algorithm
Random Forest (RF)
Neural Network (NN)
Gradient Boosting (GB)
Support Vector Machine Regressor (SVR)
Ridge Regression (RR)

2.1.2 Estimating the WCET of a target program

The WCET estimation phase for a target program is shown in Figure 2.2. 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 [117] 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 [78].

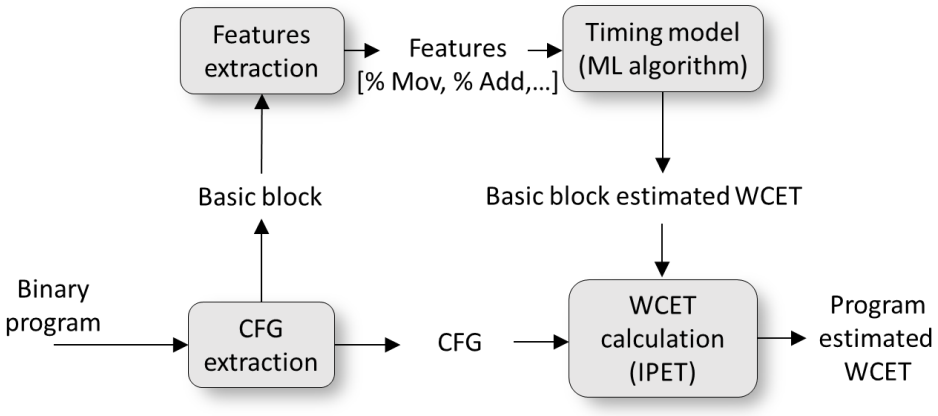

Figure 2.2 – WE-HML WCET estimation phase.

In Section 2.1.4, we show how to extend this simple formulation to take *data caches* into account.

2.1.3 Automatic generation of training data

Contrary to existing works that utilize machine learning algorithms for WCET estimation [86, 5], which often depend on a limited set of benchmarks for training, we take a more expansive approach. Drawing inspiration from [167], we employ a large dataset of automatically generated basic blocks to train the machine learning algorithms. This approach provides a diverse and comprehensive set of code snippets.

The WE-HML code generator is designed to create C source code for basic blocks using a predefined grammar. This source code is then compiled into binary. The generated basic blocks are diverse, with randomly determined numbers of statements and variables. The code incorporates all standard basic types—such as *char*, *short*, *int*, and *long*—both in their signed and unsigned variants, as well as arrays of these types. These types are selected based on user-provided proportions, ensuring diversity in the training data. Additionally, the generator includes a broad collection of common C operations, covering arithmetic and logical operations to shift-and-rotate, array indexing, and various boolean operations.

Each generated basic block starts by declaring a set of variables and subsequently performs a series of randomly selected operations on them. By design, the generated code is free of runtime exceptions, such as out-of-bounds array accesses. While the code may include *if* statements to encompass branching instructions, the generator ensures that there is no data-dependent execution. All conditional branches are designed to be predictable, with *if* statement conditions always set to *true*. It is worth noting that the WE-HML definition of a "basic block" slightly diverges from the conventional compiler domain definition. In WE-HML, a basic block can contain branching instructions to enable the timing estimation of such operations. An example of a WE-HML-generated basic block is provided in Listing 2.1. The provided C code represents a basic block that operates on an array named array_0 and various variables, encompassing conditional assignments, arithmetic operations, and bitwise shifts. Notably, all conditions, contingent upon the comparison between the variables one and zero, are met to guarantee the full execution of the code.

Listing 2.1 – 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;
  ++var_3;
  array_0[146] = -array_0[array_index];
  var_1 = array_0[array_index] * array_0[140]; }
```

The WE-HML code generator outputs C code, making it architecture-agnostic. As a result, it is suitable for WCET estimation across various processor targets. Details on the experimental conditions, implemented to minimize bias during the training of ML algorithms on automatically generated code, can be found in Section 2.2.

2.1.4 Supporting processors with data caches

The memory hierarchy significantly impacts the execution time of a basic block. When the instruction/data caches contain no information (*cold* cache), or when dirty data have to be copied back into 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 what follows, we propose a method to consider the **data caches** when estimating the

WCET of basic blocks.

Unlike traditional static cache analysis methods [123, 68, 77] that demand detailed knowledge of the cache architecture and the program's memory access patterns, WE-HML employs a learning-based approach. In this approach, data cache effects are explicitly considered during training, where the WCET of a basic block is measured and learned under an instrumented state of the cache. This allows WE-HML to more accurately estimate the cache's impact on the execution time of a basic block during the prediction phase. Specifically, WE-HML introduces the concept of cache *pollution value* to account for variable data cache conditions within a loop nest. In general, a loop nest consists of several basic blocks. When a basic block b is executed multiple times within a loop nest, its execution time can be influenced by other basic blocks (denoted as b_{others}) in the same loop nest, which can multiply the memory accesses.

During the training phase of WE-HML, we artificially simulate the memory accesses of b_{others} for each basic block b in our dataset. This is done by varying a hyperparameter called the *pollution factor* p, which is used to simulate a factor number of accesses. For the size of the accessed data, we chose arbitrarily to take the size of the data accessed by the basic block b, represented by x bytes, which we want to subject to a certain *pollution factor*. In this context, if b accesses x bytes of data, a *pollution value* of p suggests that $p * x$ bytes could be accessed within the loop nest, independently from b, potentially ejecting its data from the cache. Our observations indicate that a higher *pollution value* corresponds to an increased execution time for b.

Listing 2.2 demonstrates the code used for the execution of a basic block under the influence of cache pollution. It initiates by clearing the cache. It then repeatedly executes a basic block $BB()$, measuring its execution time within the loop controlled by nb_iter. The repetitive executions serve dual purposes:

— Capturing the processor's inherent timing variability, especially as the pollution code's introduction is *random*.

— Measuring, though without absolute guarantees, the worst-case pollution situations impacting b in loop nests.

After each execution, and as a precaution, the instruction cache and branch predictor are flushed between tests using *invalidate_brPred_icache* since its pollution effect is not currently accounted for. Finally, a function pollute(p*x) is called to simulate cache pollution by writing p*x random bytes.

Listing 2.2 – 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);
  //save execution time iteration i  = (ta - tb)
  invalidate_brPred_icache();
  pollute(p*x); // Write randomly p*x bytes
}
```

The sequence for two iterations of Listing 2.2 is shown in Figure 2.3, which offers a visual representation detailing the process of cache pollution on a basic block b, symbolized as $BB()$ when subjected to a specified pollution factor p. Initially, at the first iteration of the loop, the basic block b encounters cache misses, as illustrated in Step 1 (cold cache). Following the execution of $BB()$, data is subsequently loaded into the cache (entering warm cache state), as shown in Step 2. Transitioning to Step 3, a pollution code then simulates extra cache accesses, drawing $p * x$ bytes from an array that aligns with the L1 cache's dimensions (polluting the cache). As the process progresses to the next loop iteration in Step 4, the effects of the simulated pollution become evident. Specifically, in the Figure 2.3 example, b undergoes partial cache misses due to the disruptions caused by the simulated pollution, a phenomenon further emphasized in Step 5.

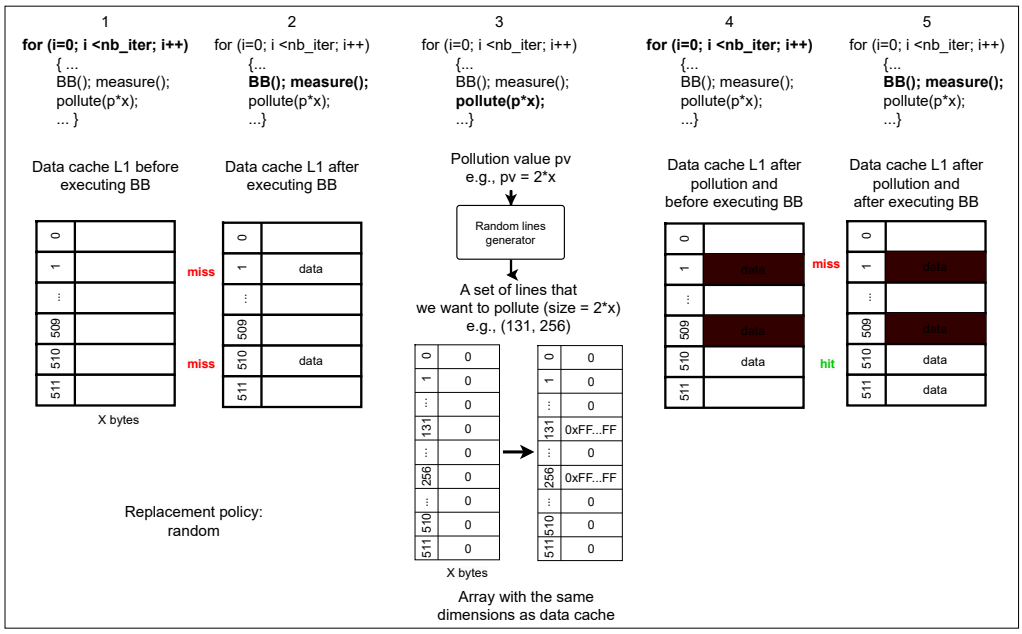

Figure 2.3 – An example illustrating data cache pollution simulation on a basic block in 5 steps. The process targets a data cache that employs a random replacement policy.

The pollution code aids in understanding the impact of cache levels on the execution time of the basic block under investigation. This understanding is achieved without the need for detailed specifications of the cache architecture—only the size of the L1 cache, which can be easily determined experimentally, is required.

Once the dataset for each basic block is prepared for every pollution factor, training of the machine learning model can start. As illustrated in the upper section of Figure 2.4, each dataset is used to train "ML model X" (machine learning algorithm X), producing multiple instances of the model, each corresponding to a specific pollution level.

During the estimation phase, depicted in the bottom section of Figure 2.4 is divided into two primary parts. The left segment, termed "estimation of maximum pollution value", outlines the process to determine *pollution values* for each basic block, starting with the control flow graph (CFG). The current approach to estimating the *pollution value* is conservative. It accounts for *all* accesses within a loop nest, meaning if there are several paths in a loop, the accesses from all paths are aggregated, ensuring safety but possibly overestimating *pollution values*. Another conservative aspect is recognizing loops as the sole source of cache reuse, overlooking reuse from function calls. Presently, cache pollution is only calculated for loop nests that consist of a function call tree with a depth greater than one. Benchmarks not adhering to this criteria are excluded. In such cases, the highest *pollution factor* is utilized.

The right segment, labeled "BB WCET Estimation", depicts the method to obtain two WCET predictions for each BB. These predictions stem from two cache conditions: cold and warm. For the cold cache scenario, the highest *pollution factor* is applied, signifying the BB's run during the initial loop iteration. In contrast, for the warm cache scenario, the *pollution factor* nearest to the statically assessed cache *pollution value* is chosen. Both these factors are incorporated into a static WCET tool to derive the program's final WCET estimate.

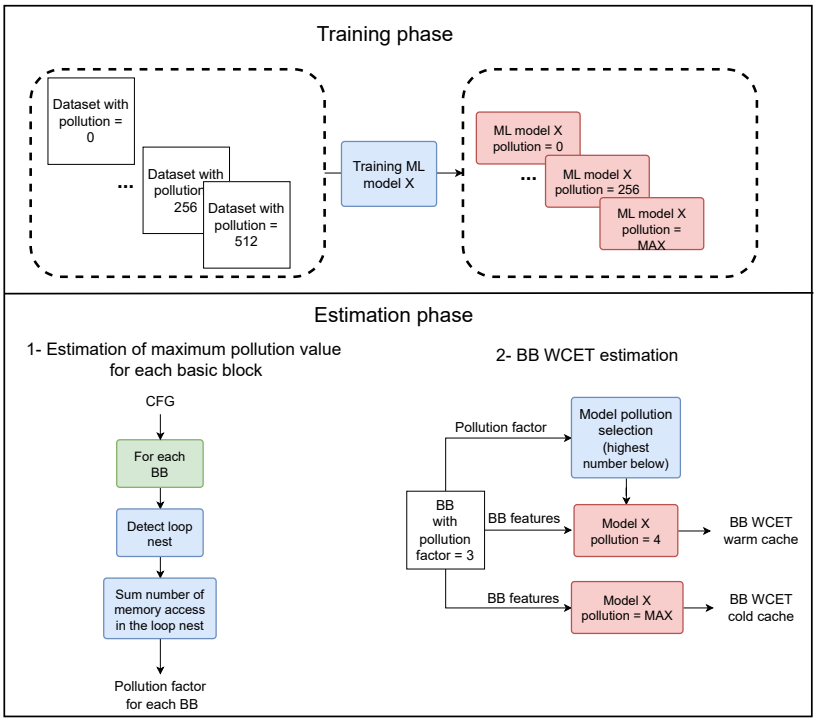

Figure 2.4 – The introduction of cache pollution during both the training and estimation (prediction) phases.

2.2 Experimental setup

In this Section, we detail the experimental setup used to evaluate WE-HML for the Raspberry Pi 3 B+ platform. The hardware and software environments are first introduced (Section 2.2.1). The programs used for evaluating the quality of predictions are presented (Section 2.2.2). We then detail the learning and prediction phases of WE-HML (Sections 2.2.3 and 2.2.4).

2.2.1 Hardware and software environments

The *Raspberry Pi 3 B+* utilizes a Broadcom BCM2837 SoC, featuring a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor with a 2-wide superscalar architecture [142]. It is equipped with a dedicated L1 cache and a 512 KiB shared L2 cache. Timing data are gathered using the processor's built-in cycle counter, accessed through the *cnt_read* function (as shown in Listing 2.2). The single-instruction requirement for reading this counter ensures negligible measurement overhead.

The device operates on Raspbian Lite, a lightweight version of the Linux operating system (Kernel version 4.19), optimized to minimize system-induced timing variations. Following the methodology of Bate et al. [16], we have configured the operating system to maintain a constant processor frequency of 800 MHz, thereby eliminating the variability

introduced by Dynamic Voltage Frequency Scaling (DVFS).

To further minimize timing noise attributable to the operating system, the code is compiled and executed as Linux kernel modules. Execution is confined to a specifically designated core (core 3), which is isolated from user tasks using the Linux *isolcpus* utility. To ensure a consistent starting point for each test, the instruction cache is invalidated and the data cache is pre-filled with dirty data, thereby establishing a cold cache state at the onset of each experiment.

2.2.2 Benchmarks

The quality of WCET predictions of programs was evaluated on 13 benchmarks from the TACLeBench benchmark suite [64]. 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 2.2.4). Table 2.2 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. Each benchmark comes with input values that exercise the longest execution path.

Table 2.2 – 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

Compiler optimizations were disabled when compiling the benchmarks to facilitate the provision of flow information during WCET analysis (if optimizations were allowed, the flow information, for example, the loop bounds, would then have to be transformed manually according to the optimizations applied by the compiler, which is error-prone [112, 45, 63]). Consideration of compiler-optimized code is left for future work.

The original code for the benchmarks *expint* and *ns* contained a long piece of code executed in one single loop iteration. As Heptane, the tool we have modified for WCET estimation, does not include any detection of such an *infeasible* path [74], 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 restructured to avoid this very long, infeasible path.

2.2.3 Implementation of the training phase

We generated a set of 15,000 basic blocks to train the machine learning (ML) algorithms. These basic blocks encompass all the instructions that the *gcc* compiler can produce for the ARM Cortex-A53 processor without optimizations. Due to the lack of official documentation on the instruction set generated by *gcc*, we validated the comprehensiveness of our basic blocks by ensuring all instructions used in benchmark tests are represented.

The training phase involved ten different pollution values, as shown in Figure 2.4. These values are powers of two, ranging from 1 to 512, yielding 10 distinct versions of each evaluated ML algorithm—one for each pollution level. The upper limit of 512 was determined empirically based on an extensive analysis of the impact of pollution on a wide array of basic blocks.

The quality of the timing model is highly dependent on the representativeness of the training data. To minimize bias, we took several precautionary measures:

— We carefully calibrated the code generator's parameters to yield a balanced mix of instruction types that closely mimic real world code without being overly specific to any particular benchmark. Additionally, we tuned the settings to produce a variety of basic block sizes, ranging from very small to considerably larger ones. Figure 2.5 represents as a box-and-whisker plot the proportion of the ARM instructions in the generated code (at the top) and benchmarks (at the bottom), respectively.

— To prevent underestimation of execution times during the training phase—a potential issue especially with branch instructions—we flushed the branch prediction tables between each timing measurement. This process is detailed in Listing 2.2. Moreover, introducing pollution code aimed to maximize basic block execution times in the presence of cache pollution within loops.

— A known limitation of our training data in this early work is that it focuses solely on the *proportions* of instructions rather than their *order of execution* within each basic block. This was a deliberate choice to expedite both the training and prediction phases.

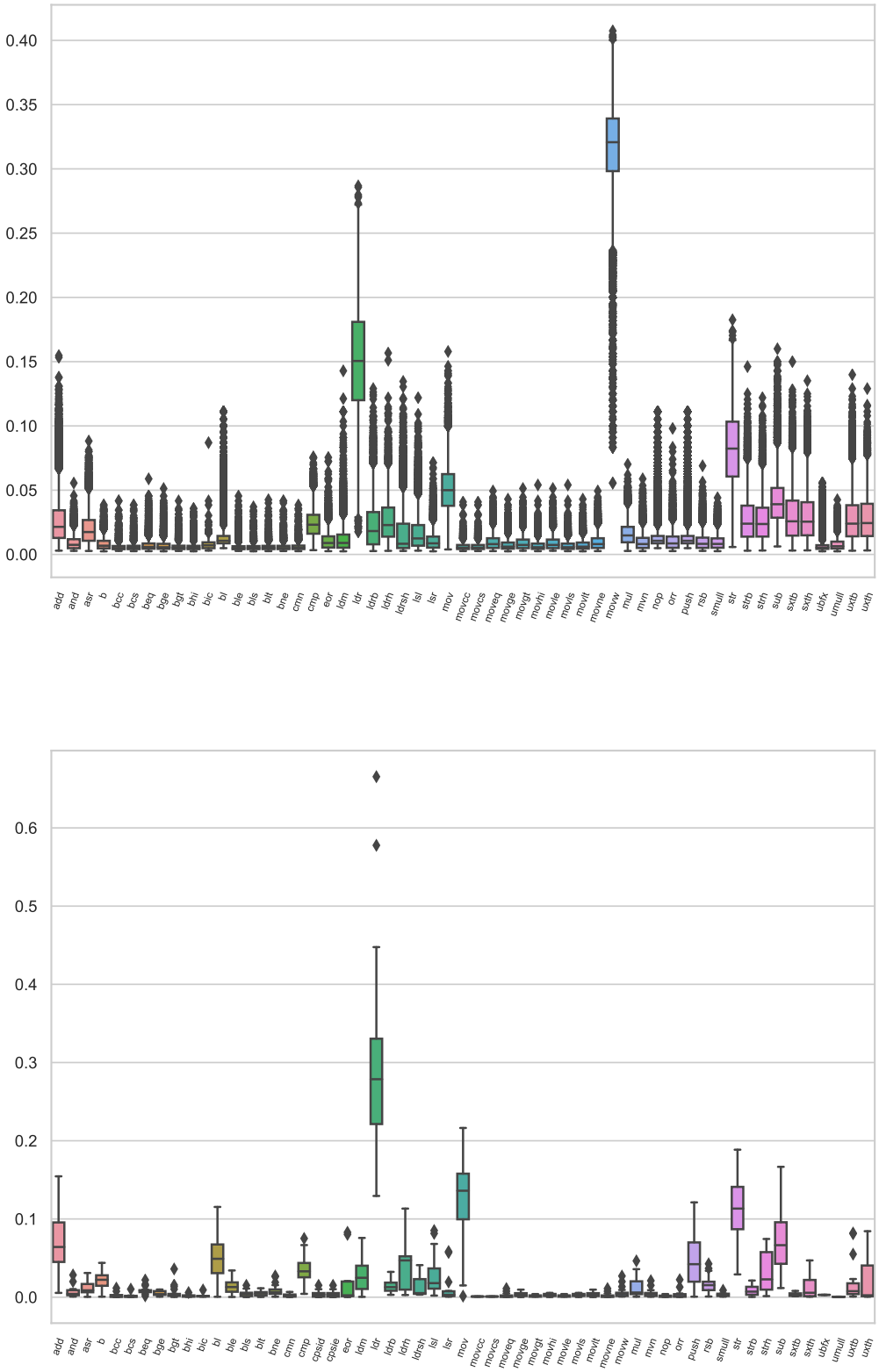

Figure 2.5 – Statistics about instruction proportion in basic blocks used for training (top), in our synthetic data and TACLeBench benchmarks (bottom).

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 is determined using measurement-based probabilistic timing analysis [29, 30]. This approach aims to exclude outliers in measurements that arise from operating system noise, targeting an exceedance probability of 10^{-3}. We employed Extreme Value Theory (EVT), specifically the Generalized Extreme Value (GEV) theory, to derive the WCET from a collection of measurements. It's crucial that the timing samples satisfy the three conditions necessary for applying EVT: stationarity, short-term independence, and long-term independence [148]. Out of the 15000 generated basic blocks, only 10000 met these three applicability conditions [144], based on a commonly-used significance level of 5%.

For both techniques, 80% of the basic blocks were used for training and cross-validation, and 20% 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 an 8-core Intel i7 processor, took around 62 minutes in total for the 5 ML algorithms.

2.2.4 Implementation of the WCET estimation phase

WCET estimation is implemented by modifying the open-source IPET-based static WCET estimation tool Heptane [78]. Heptane was modified to calculate the pollution value for each basic block. 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 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.

2.3 Experimental results

The quality of WE-HML is evaluated from different points of view. First, we evaluate the quality of the WCET predictions of entire programs (Section 2.3.1). Then we evaluate the benefit of accounting for caches (Section 2.3.2). WE-HML is then compared with a cache-agnostic measurement-based hybrid technique in Section 2.3.3. Finally, a detailed analysis of the quality of WCET predictions at the basic block level is given in Section 2.3.4.

2.3.1 Prediction of WCETs of programs

Table 2.3 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 Table 2.3 are obtained by the ML models trained with pWCET-10^{-3} values for basic blocks. 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 2.3, calculated by the less pessimistic ML technique.

Table 2.3 – Estimated WCET obtained by WE-HML versus MOET.

Benchmark	RF	NN	GB	SVR	RR	MOET	Over-estimation 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

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 *RR* (9 times). *Petrinet* is by far the most overestimated benchmark of the others. 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.

A more detailed analysis is now given for the 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 2.6 depicts the MOET and ML-predicted WCET for all selected ML algorithms.

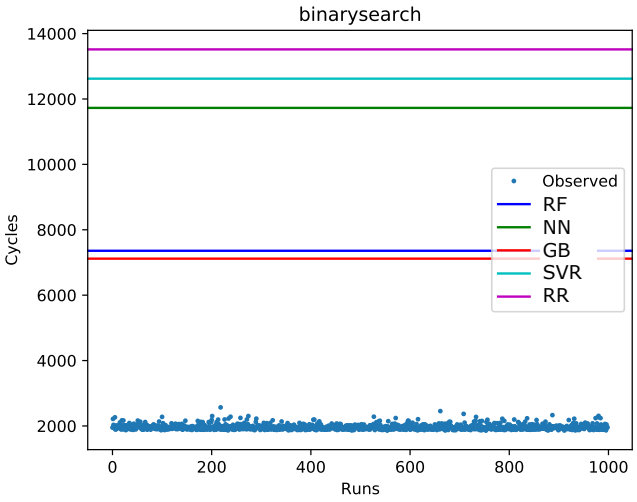

Figure 2.6 – ML-predicted WCETs versus observed execution times for *binarysearch*.

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

Using lower exceedance probability. Although we did not observe any underestimated WCET estimates using WE-HML, one may wish to change the way the WCET of basic blocks is 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.

WE-HML WCET prediction duration. 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 nonoptimized code for WCET estimation (call of a Python script for each basic block, parameter passing using files, deserialization of the ML algorithm for each basic block). We observed that RF is by far the most time-consuming ML algorithm (but also one of the most accurate ML algorithms). The other algorithms are comparatively much faster.

2.3.2 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 2.4 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 the formula $\frac{W\_nocache - WEHML}{W\_nocache}$.

Table 2.4 – Improvement (decrease) of estimated WCET resulting from cache management.

	RF	NN	GB	SVR	RR	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%

As expected, the numbers show a significant improvement brought by cache management (65% on average for all ML algorithms and all benchmarks). The *Petrinet* benchmark 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 runtime). *Petrinet* behaves similarly to *nsichneu*, with a slightly lower volume of accessed data and, therefore, a bit of cache reuse detected. The benchmarks that benefit most from cache support are the ones with deeper loop nests and a low volume of data accessed in the loop nest.

2.3.3 Comparison with a hybrid WCET estimation technique

This section compares 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 [100, 95] 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 2.3, implements edge detection in an image. Only the main (and longest to execute) basic block was instrumented to limit the cost of insertion of instrumentation.

Listing 2.3 – 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 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 [100] with instrumentation at the basic block level.

Table 2.5 – Comparison with hybrid method.

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

Table 2.5 presents a comparison of the WCET estimates obtained using WE-HML with

different machine learning algorithms and the MB-Hybrid method. For each algorithm, the table provides the WCET for the first execution time (iteration) of a basic block (BB first) and subsequent iterations (BB next). The fourth column presents the overall WCET estimated using IPET, and the final column provides the ratio $\frac{MB-Hybrid}{WCETestimated}$, which offers a comparative measure of the MB-Hybrid method's estimate against the estimates from the WE-HML.

From the results in Table 2.5, several observations can be made:

— The cache-agnostic hybrid methods, as anticipated, provide more pessimistic WCET estimates compared to the WE-HML approach. On average, the WCET estimates from these hybrid methods are approximately 2.5 times higher than those from WE-HML.

— Among the WE-HML algorithms, the Random Forest (WE-HML RF) method provides the lowest WCET estimate, while the Support Vector Regression (WE-HML SVR) method yields the highest.

It is worth noting that WE-HML offers the advantage of not requiring code instrumentation, and it effectively addresses the code coverage issue.

2.3.4 Prediction of WCETs of basic blocks

WCET prediction at the program level obviously depends on the 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 2.6 by analyzing the *Pearson correlation score* of the ML algorithm (or the coefficient of determination) as provided by Scikit-learn [139]. The higher the score, the better the prediction, with the 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 an exceedance probability of 10^{-3}, see Section 2.2.3), and then per pollution values (1, 16, 512).

Table 2.6 – Pearson correlation score of Scikit-learn ML algorithms on basic blocks, depending on the 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
SVR	0.077	0.467	0.827	0.722	0.415	0.874
RR	0.076	0.467	0.827	0.722	0.415	0.874

We observe that training the ML algorithms using the MOET of basic blocks may lead to very low Pearson correlation scores. Our examination of the training data suggests that the presence of infrequent but exceptionally high-timing outliers in the measurements could be attributed to activities from the operating system. Probabilistic techniques, such as pWCET at a threshold of 10^{-3}, are inherently more resilient to these outliers. By design, they can disregard such anomalies if they are sufficiently rare. In our observations, the learning scores across different algorithms were consistent under similar conditions. While pWCET at a threshold of 10^{-6} exhibited marginally improved Pearson correlation scores compared to pWCET at 10^{-3}, it resulted in considerably higher WCETs—averaging a 35% increase across 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.

2.4 Conclusion

In this chapter, we introduced a novel hybrid WCET estimation technique called *WE-HML*. This approach leverages machine learning (ML) algorithms to estimate the WCET of basic blocks, thus mitigating the need for in-depth architectural knowledge. Unlike existing ML-based WCET estimation methods, *WE-HML* takes into account data caches and operates directly at the machine code level. The experimental results reveal that, while no single ML algorithm consistently outperforms others across all benchmarks, *WE-HML* generates WCET predictions that consistently exceed actual observed execution times. Moreover, the incorporation of cache modeling improves the average accuracy of our WCET estimates by 65% when compared to a cache-agnostic counterpart. It is worth noting that the majority of WCET estimates for the tested benchmarks fall within a range of seconds.

While the initial results are promising, several avenues for improvement exist:

Oversimplified code characterization. The current feature set used for learning and predicting WCET overlooks critical timing-related information derived from machine code (e.g., dependencies between registers and data access). This lack of consideration compromises the model's ability to provide timing estimates that accurately capture pipeline behavior and cache effects, especially those related to spatial locality. Moreover,

the machine learning algorithms in use are not naturally equipped to account for the sequential order of machine instructions within a basic block, indicating a need for a more refined approach.

Dissimilarity in basic block sizes. The sizes of basic blocks used during the *training phase* are not consistent with those encountered during the *prediction phase*. This poses a challenge to the precision of *WE-HML*. Our choice of longer instruction sequences during training was a strategy to offset measurement noise stemming from our reliance on software-based measurements. Fortunately, the integration of real world basic blocks extracted from program benchmarks, coupled with advanced hardware measurement methods such as JTAG [152], could solve these issues and enhance the accuracy and reliability of our technique.

In the following chapters, we will investigate advanced ML techniques that can effectively account for instruction dependencies. We will also employ hardware measurement solutions on real programs to extract execution times for actual basic block sizes while minimizing probe effects.

Chapter 3

ACET Estimation: A Dive into LSTM and Transformers

Recent research has leveraged machine learning to estimate the worst-case execution time (WCET) of software [88, 91, 22, 7]. Although these approaches address early-stage WCET estimations and code coverage challenges [111], they often overlook instruction sequencing and register dependencies, leading to overly pessimistic predictions.

We aim to improve the accuracy of machine learning models by considering not only the dependencies between instructions within a basic block but also the *context* created by code executed before the BB. In this chapter, our focus is directed toward the average-case execution time (ACET) of basic blocks (BB).

We present two new models, *CATREEN* and *ORXESTRA*. *CATREEN*, which stands for "Context-Aware code Timing estimation with stacked REcurrEnt Networks", leverages the power of stacked LSTM [82] layers, capitalizing on their ability to capture long-term sequential dependencies. This makes it adept at understanding the relationships between instruction sequences. On the other hand, *ORXESTRA*, which stands for "cOntext-awaRe eXEcution Time eStimation using TRAnsformers", employs the Transformers XL [43] architecture. The Transformers XL is renowned for its self-attention mechanism, enabling it to discern the significance of different parts of an input sequence, thus offering a precise understanding of the context.

Both *CATREEN* and *ORXESTRA* estimate the execution time of a basic block b by focusing on its preceding sequence of basic blocks. This approach ensures that the models account for the hardware's state, including the pipeline, cache hierarchy, and branch prediction, providing a comprehensive understanding of the execution dynamics influenced by prior instructions.

CATREEN has been published at ICTAI 2022:

"**Abderaouf N., AMALOU**, Elisa FROMONT, and Isabelle PUAUT. "CATREEN: Context-Aware Code Timing Estimation with Stacked Recurrent Networks." The 34th IEEE International Conference on Tools with Artificial Intelligence (ICTAI) IEEE, 2022."

Hypotheses

We outline the assumptions and conditions under which our proposed models are evaluated and operated.

— We consider the execution context size as a hyperparameter and limit our focus to a specific number of basic blocks. This constraint on the number of basic blocks is intentional, as including all executed basic blocks could add unnecessary complexity and computational overhead without substantially enhancing the accuracy of our timing estimates.

— This chapter's estimation phase concentrates on execution scenarios where the sequence of executed basic blocks is known, meaning the context is predefined for specific inputs. For more general scenarios, where the execution trace is typically not predetermined, the discussion is deferred to the final chapter 4. We will introduce a methodology for determining the execution context (sequence of BBs) in worst-case scenarios to estimate the WCET.

The structure of this chapter is outlined as follows. Section 3.1 offers an in-depth presentation of our machine learning models, focusing on their neural architecture details. Section 3.2 describes the baselines, the training dataset, and the experimental methodology for evaluating LSTM-based and Transformers-based timing models. A comparative performance evaluation against existing state-of-the-art techniques is elaborated in Section 3.3. Finally, Section 3.4 provides a conclusive analysis of our findings.

3.1 New machine learning architectures for timing estimation

This section provides motivation for context awareness Subsection 3.1.1 and in-depth details on machine learning architectures used for average time estimation. First, the LSTM-based model called CATREEN is detailed in Subsection 3.1.2, followed by the solution that employs Transformers XL called ORXESTRA architectures 3.1.3.

3.1.1 Motivation for context awerness

The primary reason for the need to consider execution context lies in the increased complexity of modern processors. This complexity, arising mainly from integrating various hardware accelerators designed to reduce program execution time, renders machine learning timing models based solely on extracted static features simplistic and prone to error.

The cohabitation of these components makes the timing modeling of a processor difficult and sometimes unpredictable. Let us take, for example, a processor architecture with an N-stage pipeline, data and instruction caches, and a branch predictor. The **pipeline** allows the execution of a new instruction to start without waiting for the previous one to finish as long as there are no register dependencies. The **cache memory** is a small, fast memory that stores copies of instructions/data from frequently used locations in the main memory. Finally, the **branch predictor** allows the processor to speculate about the outcome of a conditional branch (e.g., fill in the pipeline with the instructions of the predicted branch direction) to accelerate the execution.

These components introduce dependencies between successive instructions, making the timing of a sequence of instructions dependent on its *execution context*. This is illustrated in Figure 3.1, the left top square represents a C code snippet delimited by two *comments* //Start and //End, and the right boxes represent the machine instructions after compilation, separated into *basic blocks* (BB). The binary code in our example is made of 3 BBs denoted in the figure as BB1, BB2, and BB3.

Regarding the effect of **pipelines**, the execution time of a BB depends on the BB executed before, which defines the level of parallelism between the successive BBs. For example, the execution time of BB3 must consider that BB2 has been executed before. Plain arrows in Figure 3.1 show all pairs of BBs concerned by this effect.

As far as the **branch prediction** is concerned, if the branch predictor successfully predicts the outcome of the branch instruction contained in BB1 (the "bgt" instruction, which means: branch if i is greater than 9), the first instructions of BB2 are executed before the result of the comparison is known. This makes the timing of BB2 dependent on the previous execution of BB1. Similarly, in the case of an incorrect branch prediction, the timing of BB4 depends on BB1. Dashed arrows depict these effects.

Regarding the effect of **caches** on our example, if the variable i is loaded in the cache in basic block BB1 and is not evicted from the cache when BB3 is executed, the execution time of BB3 is reduced, making the timing of BB3 dependent on the previous execution of BB1. Dotted arrows depict these cache-related effects in the figure.

Precisely estimating the timing of basic blocks requires accounting for all these intra- and inter-BB dependencies.

Building upon existing work such as ITHEMAL [129], which utilizes hierarchical multiscale Long Short-Term Memory [82] (LSTM) layers to predict the throughput of isolated basic blocks (BBs), we introduce two machine learning models specifically designed for context-aware code timing estimation *CATREEN* and *ORXESTRA*.

Unlike ITHEMAL, which isolates each BB using a dynamic binary instrumentation tool [27] and repetitively executes it to measure its best-case performance, our models

estimate the execution time of BBs within their respective "execution contexts." Specifically, we consider the sequence of instructions executed immediately before the BB under study. This context-aware approach offers a more accurate reflection of a BB's actual execution time, as it captures timing nuances that arise from hardware-software interactions in sequences extending beyond individual BBs.

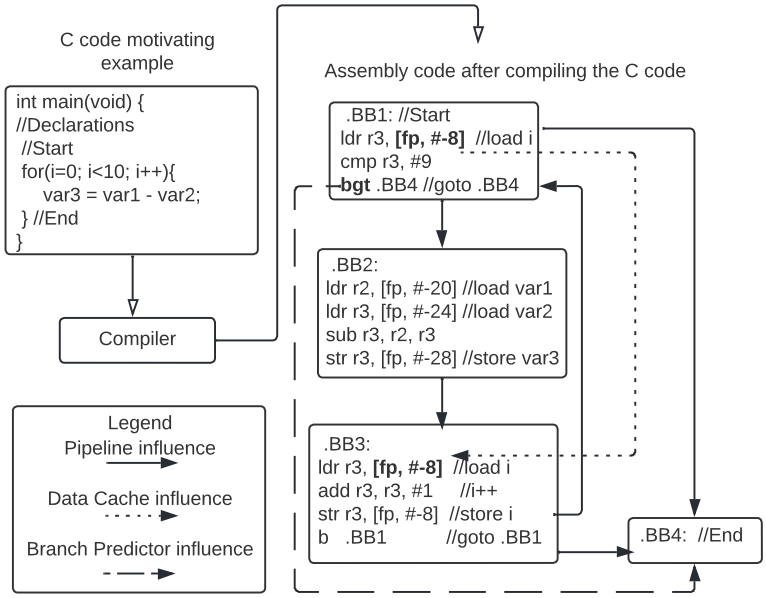

Figure 3.1 – Inter Basic Blocks hardware dependencies.

3.1.2 ACET estimation using LSTMs, CATREEN

CATREEN leverages a stacked recurrent neural network architecture to predict the execution time of a basic block b (the *basic block under analysis*) given its execution context (sequence of basic blocks executed before b). By considering the execution history of basic blocks, CATREEN naturally accounts for the state of the hardware when executing the basic block (pipeline, cache hierarchy, and branch prediction).

Long Short-Term Memory [82] (LSTM) is a special kind of recurrent neural network, capable of learning long-term dependencies [83] in the sequences. LSTM architectures have mechanisms called gates that are trained to choose whether or not to keep particular sequential information (more details about LSTM's inner functions can be found on page 47). These architectures are, for example, extensively used in domains such as natural language processing [168] [147], where the context of a word in a sentence is useful for learning its characteristics. Figure 3.2 represents the overall architecture of CATREEN. CATREEN estimates the execution time of a basic block within a sequence of basic blocks

in a similar way that an LSTM would process a paragraph in natural language to predict, for example, the sentiment (positive or negative) of a given sentence in this paragraph. The analogy is as follows: an instruction represents a word. An instruction comprises an opcode and one or more operands, which can be treated as letters that constitute this word. A set of instructions will form a basic block (a sentence), and a sequence of basic blocks will represent a paragraph.

Architecture of CATREEN. The architecture of CATREEN comprises five layers, depicted in Figure 3.2 from top to bottom. The first layer is a *tokenization and embedding* layer that preprocesses the assembly language, extracted from the machine code and generates inputs that are understandable by the next layers. The next three layers are the central layers of CATREEN. They are LSTM layers: one to process an instruction (Instruction Layer in the Figure), one to process a basic block (BB layer in the figure), and one to process a sequence of basic blocks (Sequence Layer in the figure). Finally, we create a sort of weak connection to save the basic block under analysis (last BB in the sequence) and concatenate it with the formed context, and send it to a dense layer that is used to predict the final timing output (timing a basic block in the context of the previously executed basic blocks). More details on each layer are given below.

Data preparation for CATREEN (tokenziation). In order for the LSTM to properly interpret basic blocks, an encoding of machine code into a sequence of integers is needed, where each integer represents an index (token) in a predefined dictionary. This is performed as follows. We preprocess the raw data (assembly code) by encoding each basic element of an instruction (opcode and operand) with a unique number (token), with a special treatment given to the operands:

— All immediate operands (constants) are encoded with the same token value.

— All address operands are encoded with the same token value.

— Addressing modes using registers are differentiated by assigning a distinct token value per pair (register number, addressing mode) with the addressing modes supported by the considered architecture (e.g., for the ARM targets, direct access, indirect access, and indirect access with offset).

The reason for implementing these rules is to reduce the sequence length, given that we know LSTM models struggle with processing very long sequences [97].

For example, consider the following sequence of ARM instructions:

MOVEQ R3, #0x0 ; LDR R3, [R3] ; BL 0x080008DC ; STR R3, [R3, #0x0166]; MOVGT R3, #0x0230 ;

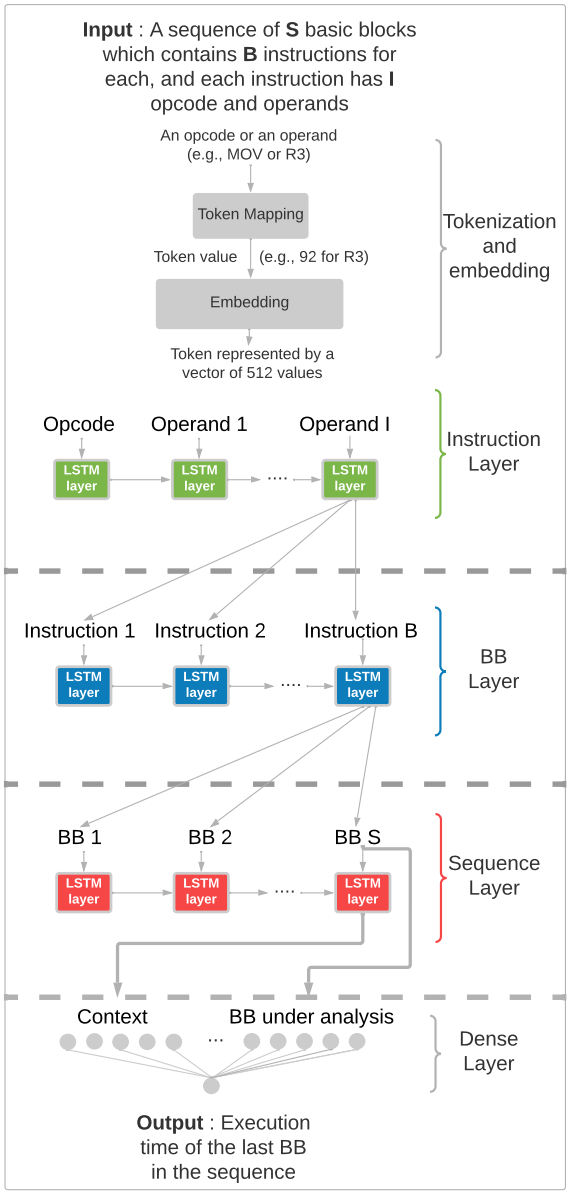

Figure 3.2 – Architecture of CATREEN. The input is a sequence of basic blocks consisting of a sequence of instructions, which are themselves sequences of operands/opcodes. The (grey) upper part of the figure shows the processing of one such operand/opcode. CATREEN calculates (lower part) a timing estimation for the last basic block in the input sequence.

The final output of the encoding for the considered sequence of instructions is:*[19, 500, 380, 999, 28, 500, 501, 999, 65, 381, 999, 35, 500, 502, 999, 20, 500, 999]*. Where 999 is a separator between instructions.

In order to be suitable inputs for LSTM layers, each token is again encoded into a distinct fixed-size vector of floats within the interval [-1,1] using word2vec [133].

LSTM layers of CATREEN. Since we do not, *a priori*, know what is the length

of the sequences (i.e., how many basic blocks they contain) nor the basic blocks length (i.e., how many instructions they contain) nor even the length of the instructions (i.e., how many operands they contain), we model these data with LSTM, which are suited to variable-length sequences. Three levels of LSTM are used for that purpose:

— The *Instruction Layer* (in green in figure 3.2) is the first LSTM layer in CATREEN. It takes as input a sequence of embedded code operands/opcodes (i.e., an instruction). We loosely denote the length of the sequence of operands/opcodes by O (resp. I and B in the subsequent layers) even though it differs from one instruction to another. Each LSTM layer processes the entire sequence of embedded tokens from an instruction (indexed by $t \in [0..O-1]$) and produces a single final output representation at $t = O - 1$ for the entire instruction. The outputs (instructions) are treated one by one by the next LSTM layer (BB layer, for the Basic Block layer). We loosely denote the dimension per layer by N, even though it may differ for the three LSTM layers. The selected hidden size per layer is given in Section 3.3.2.

— The *BB layer* (in blue in figure 3.2) processes the sequence (of length I) of embedded instructions in a basic block and produces a representation for each basic block. Again, all representations of a basic block are treated one element by one by the next LSTM layer (Sequence layer).

— Finally, the *Sequence Layer* (in red in figure 3.2) processes the sequence of S basic blocks within a sequence. Similarly to the other LSTM layers, it produces at the end a representation of the sequence or what we call *Context*. With the value of *Context* concatenated to the value of the *BB under analysis* (that we save from the previous layer *BBs*) are then given as inputs to a fully connected linear layer connected to a single output which produces an estimate of the execution time of the last basic block of the sequence given the execution history.

3.1.3 ACET estimation using Transformers, ORXESTRA

Our second model, ORXESTRA, uses Transformers XL (TXL) [44] to predict performance. Transformers, introduced in [165], are machine learning neural architectures initially developed for natural language processing (e.g., language translation, text summarization, text-to-speech, ...). They use self-attention mechanisms that allow the model to learn how to weigh different parts of the (potentially sequential) input data. This weighting scheme, in turn, allows the model to capture dependencies between the elements of a sequence (e.g., it allows the model to take into account contexts). The original Transformers architecture [164] has a fixed-length context window (as explained in 1.3.2 and may struggle with sequential data containing long-term dependencies. Dai et al. then

introduced a variation called Transformers XL [44] to address this limitation. The TXL architecture (detailed in 1.3.2) uses *memory-augmented attention* to retain better and make use of the information from earlier in the sequence. This makes it more suitable than the original Transformers for handling assembly code, which often involves long code sequences, and thus, to account for long-term dependencies between the code elements in order to perform our timing estimation task.

Architecture of ORXESTRA

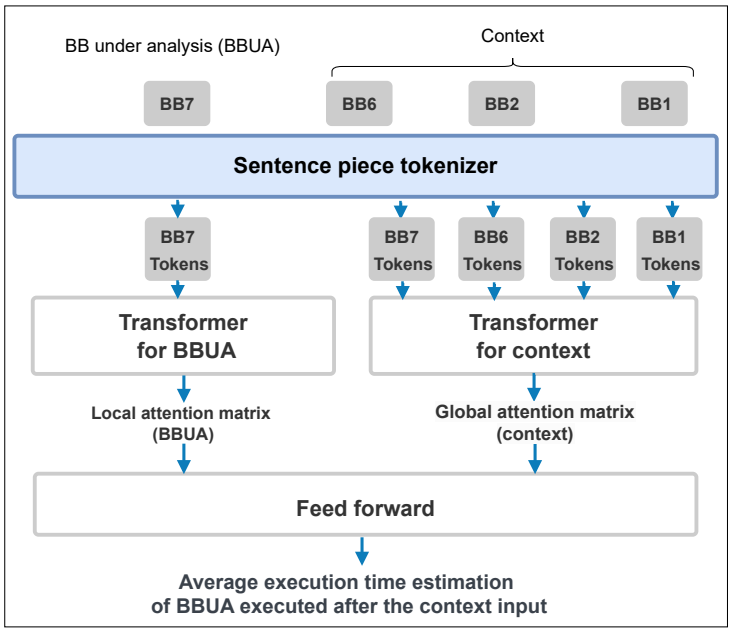

Figure 3.3 – ORXESTRA Transformers XL-based architecture.

ORXESTRA's architecture, as depicted in Figure 3.3, consists of several components.

Tokenization. At ORXESTRA core, it utilizes a tokenizer called *Sentencepiece* [103]. The *Sentencepiece* algorithm progressively merges characters and sequences of characters, thereby generating a vocabulary of subword units based on observed statistical patterns. To illustrate, let's consider the same example as in Subsection 3.1.2:

> *MOVEQ R3, #0x0 ; LDR R3, [R3] ; BL 0x080008DC ; STR R3, [R3, #0x0166]; MOVGT R3, #0x0230 ;*

After applying *Sentencepiece*, the sequence is segmented into the following subwords:

> *MOV_ EQ R3, #0x_ 0 LDR R3 [R3] BL 0x_ 0800_ 08DC STR R3 [R3_ #0x_ 0166] MOV_ GT R3 #0x_ 0230*

Although this new sequence may be longer than one derived using a fixed vocabulary, it effectively addresses the challenge of out-of-vocabulary (OOV)[122] [1]. In contrast to using a fixed dictionary, which creates unknown tokens for newly encountered words, *Sentencepiece* ensures that the binary programs written in the target instruction set will always be able to tokenize it, enabling efficient processing by the Transformers. As the tokenization generated by Sentencepiece is longer, it severely disadvantages LSTM models, which cannot process sequences longer than 200 tokens at a time (as empirically demonstrated in the work described in [97]). Therefore, for CATREEN, we prefer to stick with traditional tokenization (fixed dictionary), adding any missing words to the dictionary when encountered, while for ORXESTRA, we will use Sentencepiece to be able to account for addresses and access memory.

Attention matrix generation. The entire ORXESTRA's system uses two Transformers XL, each serving a specific purpose. The first Transformers, displayed on the right side of the figure, focuses on creating an embedding representation of the execution context. The second Transformers, depicted on the left side of the figure, takes care of processing the basic block under analysis (BBUA), as it holds the main information to predict the execution time.

When these two Transformers operate, they produce an attention matrix (i.e., a weighing matrix of size $\#token \times \#token$) with a fixed size of $\#token = 512$. Essentially, when a context or a BBUA is presented as input, it is divided into smaller segments of 512 tokens. Thanks to the TXL memory mechanism, the system retains information from the previously processed 512 tokens while continuing to process longer sequences. Padding is applied at the beginning of each token sequence in order to obtain a sequence with a size that is a multiple of 512. ORXESTRA generates an attention matrix when processing the context (called here the *global attention matrix*), and a *local attention matrix* for the BBUA. These two matrices are concatenated and provided as input to a subsequent feedforward (multi-layer perceptron) neural network (top of the figure). This network estimates the final average execution time (AET) associated with the BBUA.

Training of ORXESTRA

Training ORXESTRA consists of two main stages: pretraining each TXL (in practice, the same pretrained model is used twice for the two TXL) and finetuning the entire architecture. CATREEN, on the other hand, is only finetuned.

Pretraining. During the pretraining phase, the TXL is first trained using a self-supervised learning approach. Self-supervised pretraining is a technique often used in

1. OOV in machine learning can result in issues such as reduced generalization, information loss, cascading errors, and domain adaptation challenges

machine learning to leverage unlabeled data in order to "initialize" the weights of an (often large) neural network and help it learn better data representation to solve the final task. Training for this final task often involves a much smaller set of labeled data than the number of data used during the pretraining phase. A supervised learning task is designed from the unlabeled data. For example, in NLP, a standard pretext task consists of predicting a (randomly chosen) "masked" word in a sequence, also called masked language modeling (MLM) as explained with example in 51 of this document. In our working context, our goal for this pretraining phase is to enable the model to understand the structure of assembly instructions presented in a textual format. This is achieved by masking random operations or operands within the instruction sequence and training the model to predict them as output in a self-supervised training scheme. By doing so, we can leverage a very large (unlabeled) dataset consisting of thousands of disassembled binary programs to learn a rich representation of assembly code (i.e., to have a relevant initialization of the weights of our TXL) that can be used for the final training phase.

Finetuning. In the finetuning stage, ORXESTRA is trained to predict the execution time of individual BB in context. This phase employs a specific set of programs, a target processor, and a measurement tool. The execution times of basic blocks are measured (to obtain the training labels), and the corresponding instruction sequences are tokenized. To construct the training dataset, each BB's median execution time is considered, along with the tokenized BB itself and its associated context. The context size, which represents the number of basic blocks, serves as a hyperparameter.

3.2 Experimental setup

The experimental setup for this chapter begins with Section 3.2.1, where we discuss the dataset sources and their compositions that are instrumental in training and evaluating our two proposed machine learning models, CATREEN and ORXESTRA. Following this, Section 3.2.2 provides insights into the baseline methodologies that serve as benchmarks for comparing the effectiveness and performance of our models. Section 3.2.3 outlines the hardware and software configurations used. Finally, Section 3.2.4 delves into the details of our learning setup, including a list of relevant hyperparameters.

3.2.1 Datasets and benchmarks

As stated before, training ORXESTRA involves two primary steps: pretraining and finetuning. In the pretraining phase, which learns the structure of machine code, a large

dataset of programs from CodeNet [140] is used. This dataset contains about 900,000 C programs obtained from public submissions on competitive programming websites. These programs undergo cross-compilation to the target architecture (using the -O0, -O1, -O2, or -O3 optimization option randomly) and are disassembled using the GNU binary tool *objdump*. We extract relevant information such as addresses and identification of BBs from the textual output of *objdump*. It is important to note that the programs themselves are not executed during this process; they are used as a form of natural language data. The pretraining dataset is further utilized by *Sentencepiece*, which undergoes its own learning phase to produce a model capable of tokenizing our data.

To finetune ORXESTRA and train CATREEN, a varied set of publicly available programs is employed, namely *The Algorithms*[2], *MiBench* [75], and *Polybench* [174]. Basic blocks and their respective contexts are extracted from these programs with a few modifications to obtain relevant data. For example, all instances of *printf* and system calls, which introduce redundant BBs and can potentially lead to creating an unbalanced dataset and overfitting problems during training, are eliminated from these programs. In Table 3.1, a summary of each benchmark suite is provided, including the number of programs in each dataset and the total count of unique basic blocks[3] encountered during the execution of each program.

Ground truth timing generation

To acquire the timing values (labels) required for the finetuning phase of CATREEN and ORXESTRA, we utilize either a hardware-based or a software-based approach, depending on their availability. The hardware solution is preferred when available, owing to its minimal interference with execution (known as the "probe effect"). In both cases, we conduct 1000 execution time measurements for each basic block, preceded by cache warming (achieved by running the program 20 times).

Hardware-based timing instrumentation uses the Joint Test Action Group (JTAG) interface for the hardware solution and utilizes the J-Trace Pro trace solution from Segger [152]. This allows us to connect to the JTAG interface of the target processors, specifically the Cortex-M4 and Cortex-M7 in our case. To generate execution traces, we use Ozone [72], a cross-platform debugger and performance analyzer, in conjunction with J-Trace Pro. These traces provide valuable information such as the cycle counter value, instruction address, opcode, operands, and corresponding assembly code for each instruction. Figure 3.4 shows a snippet of an execution trace extracted using the tool.

2. Available here: https://github.com/TheAlgorithms/C

3. By "unique basic blocks," we mean a basic block that is unique in both its context and its composition.

Cycle Count	Address	Opcode	Operands
65 899 679	080002E6	LDR.W	R2, [SP, #0x0804]
65 899 681	080002EA	LDR.W	R3, [SP, #0x0800]
65 899 683	080002EE	CMP	R2, R3
65 899 685	080002F0	BEQ	0x0800033E ; <main>+0xA6
65 899 687	080002F2	LDR.W	R3, [SP, #0x0824]
65 899 689	080002F6	ADDS	R2, R3, #2
65 899 691	080002F8	ADDW	R3, SP, #0x0828
65 899 693	080002FC	SUBW	R3, R3, #0x0828
65 899 695	08000300	LDR.W	R3, [R3, R2, LSL #2]
65 899 697	08000304	LDR.W	R2, [SP, #0x0808]
65 899 699	08000308	SUBS	R1, R2, R3
65 899 701	0800030A	ADDW	R3, SP, #0x0828
65 899 703	0800030E	SUB.W	R3, R3, #0x0428
65 899 705	08000312	LDR.W	R2, [SP, #0x0824]
65 899 707	08000316	STR.W	R1, [R3, R2, LSL #2]

Figure 3.4 – Example of an execution trace, extracted from OZONE tool [72].

The software-based approach employs code instrumentation to measure the execution time (reading the cycle counter register before and after executing a basic block as it was done in Chapiter 2 page 67) of individual basic blocks within a program. For each basic block, annotations are used to obtain maximum loop iteration, allowing us to map the execution time with the iteration number automatically. The execution trace, along with the corresponding assembly code for the timed basic block, is acquired using the GNU Debugger (GDB), where we unroll the whole program and save each executed assembly instruction.

Data preprocessing

To generate the final dataset, the execution trace is processed in several steps. First, to mitigate the impact of outliers, the median value of multiple execution timings for each BB is used as the ground truth timing. Both *CATREEN* and *ORXESTRA* are trained using a normalized timing value, calculated as this median value divided by the number of instructions in the BB. Subsequently, sequences of BBs are assembled for training; the prefix sequence leading up to the last BB represents its context. For instance, in *CATREEN*, if a sequence consists of [BB0, BB1, BB2, BB3], BB3 is the basic block under analysis, while [BB0, BB1, BB2] serves as its context. In the case of *ORXESTRA*, the context generation needs the basic block under analysis, so we add it into its context, which will be, in this case, formed by [BB0, BB1, BB2, BB3]. Then, tokenization is performed for each model.

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 [174]	A collection of benchmarks containing static control parts. The purpose is to uniformize the execution and monitoring of kernels	30	11224
MiBench [75]	A free, commercially representative embedded benchmark suite	14	8324
Total	-	244	31671

Table 3.1 – Composition of the dataset for the finetuning phase, showing benchmarks, each accompanied by a brief description, the number of programs, and the total count of basic blocks retrieved per program. This dataset serves as the training and testing of all competitors also.

3.2.2 Baselines

CATREEN and ORXESTRA are compared to three context-agnostic timing predictors.

The first context-agnostic competitor is a Multi-Layer Perceptron (MLP) regressor, loosely referred to as a Neural Network (NN) similar to the one used in WE-HML in Chapter 2. Although not a naive approach, the neural network follows a feed-forward architecture that does not incorporate context information and further requires a fixed-size input. Our NN implementation uses 233 static features of the basic blocks as input, basically the proportion of different machine instruction types (e.g., MOV, ADD, LDR). We use a greedy search algorithm to determine the optimal hyperparameters for the NN, including its number of hidden layers, the optimizer, the learning rate, and the loss function. Based on the validation dataset, the best parameters are: hidden layer sizes set to $512, 256, 128$; learning rate set to "adaptive" with initialization at 0.001, and use of "adam" solver. These hyper-parameters are coherent with what was used in [7].

Our second context-agnostic baseline is ITHEMAL [129] (we explained how it works in detail on page 59), which uses LSTMs for execution time prediction. We re-implemented ITHEMAL from the original paper, porting the tokenization and embedding step of ITHEMAL to the ARM instruction set. Additionally, we created a tuned version of the model's hyperparameters that we used during experimentation to fit the new data better.

The third context-agnostic baseline is a re-implementation of BERT [54] (the encoder of a Transformers), here called "Transformers vanilla". The implementation is based on the

work PALMTREE [116], which demonstrates the superiority of the BERT Transformers to learn how to represent a basic block compared to other embedding like word2vec [133], instruciton2vec [110] and asm2vec [58]. This approach involves pretraining BERT, similarly to ORXESTRA, using the masked language modeling task specifically designed for ARM assembly code. It takes a single basic block as input and predicts its timing hence without considering the execution context information. Our objective with this competitor is to compare ORXESTRA with a Transformers model that performs similarly to ITHEMAL to assess the influence of context awareness on the same type of neural architecture.

In Table 3.2, we present a summary of the architecture hyperparameters used by all competing models. To facilitate a fair comparison, the context size parameter is kept consistent for both *ORXESTRA* and *CATREEN*. The hyperparameters for *CATREEN* and ITHEMAL were chosen based on the specifications provided in ITHEMAL's original paper [129]. Similarly, the parameters for *ORXESTRA* and Transformers vanilla were primarily influenced by the PALMTREE study [116].

Hyperparameter	ITHEMAL	CATREEN	Transformers vanilla	ORXESTRA
Embedding size	512	512	512	512
Feed forward structure and size	128	256, 256	512, 256, 128	512, 256, 128
Number of layers	2 LSTMs	3 LSTMs	6	4
Number of attention head	NA	NA	8	4
Memory length	NA	NA	NA	1024

Table 3.2 – Hyperparameters for deep learning architectures, including ITHEMAL, *CATREEN*, Transformers vanilla, and *ORXESTRA*, are presented. (NA: Not Applicable).

The loss function, optimizer, and learning rate parameters will be finetuned and discussed in Subsection 3.3.2

3.2.3 Hardware and software setups

Our experiments encompass a variety of Arm processors, summarized in Table 3.3. The Cortex-M4 processor features a simple in-order pipeline with three stages and no cache. This processor enables us to validate our method on a deterministic processor with precise timing measurements obtained through the JTAG interface. The more advanced Cortex-M7 processor possesses a 6-stage in-order pipeline, data and instruction caches, and a branch predictor. The Cortex-A53 processor, hosted in a Raspberry Pi 3 features an 8-stage in-order pipeline, two levels of data and instruction caches, and a branch predictor. The Cortex-A72 processor, hosted in a Raspberry Pi 4 differs from the A53 through its out-

of-order pipeline. Since the Cortex-A53 and Cortex-A72 lack a JTAG interface, we rely on reading the cycle counter register for timing measurements as explained in Section 3.2.1.

Table 3.3 – Summary of the processors used and their micro-architectural features.

Target	M4	M7	A53	A72
Measurement tool	JTAG	JTAG	Software	Software
OS?	Baremetal	Baremetal	Linux	Linux
Pipeline/#stages	In-order/3	In-order/6	In-order/8	Out-of-order
Branch predictor	No	Yes	Yes	Yes
Cache memory	No	L1	L2	L2
Replacement policy	No	Random	Random	Random

3.2.4 Setup for the learning phase

PyTorch was used to implement our model and the baseline ones. ORXESTRA was trained on a Tesla V100 GPU. Each setting target (processor) required two days for ORXESTRA training: 1,5 days for pretraining and 0,5 days to finetune the model. The perplexity[33] [4] score was chosen as the value to optimize during pretraining (see Equation 3.1). All the datasets (even in the pretraining phase) are split into training (70%), validation (10%), and test (the rest) sets containing different BBs. The MAPE (Mean Absolute Percentage Error, see Equation 40) is used to assess the performance of each model. It evaluates how far (as a percentage) the prediction is from the true timing.

$$\text{Perplexity}(D) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i)}} \tag{3.1}$$

Where:

— N is the total number of events or words in the dataset.

— w_i represents the ith event or word in the dataset.

— $P(w_i)$ is the probability assigned to event w_i by the probability model. It is the estimated likelihood of observing event w_i based on the model.

3.3 Experimental results

Following standard evaluation methodologies for language models, —ORXESTRA undergoes two main types of evaluations: *intrinsic evaluation* and *extrinsic evaluation*.

4. Perplexity is a measure of how well a probability model predicts a sample or a sequence of events. A lower perplexity indicates a better model fit to the data.

For ORXESTRA, the intrinsic evaluation is conducted right after the pretraining phase. This stage involves assessing the model using specific unsupervised learning metrics, in our case, the perplexity score. The findings of this initial evaluation are presented in Section 3.3.1. CATREEN, on the other hand, does not undergo a pretraining phase and is, therefore not part of this intrinsic evaluation.

Before delving into extrinsic evaluation, an essential step of hyperparameter tuning is carried out for both models. Here, we finetune hyperparameters such as the loss function, learning rate, and optimizer. The details of this tuning process are outlined in Section 3.3.2.

Subsequently, extrinsic evaluation is conducted for both ORXESTRA and CATREEN. This part of the evaluation focuses on their performance in real world tasks, specifically predicting BB average execution times. The mean absolute percentage error metric serves as the key performance indicator, and results are detailed in Section 3.3.3.

Further analyses are performed to examine the impact of various factors on the models:

— The effect of context size on timing estimates for both models is discussed in Section 3.3.4.

— The scalability of the models, particularly when estimating timing for large basic blocks, is examined in Section 3.3.5.

— The influence of code optimization techniques on timing estimations is explored in Section 3.3.6.

Lastly, Section 3.3.7 provides insights into the processing throughput for each model. This section offers estimates on the number of instructions that can be processed per second for the purpose of timing estimation.

3.3.1 Evaluation of the pretraining (for ORXESTRA and Transformers vanilla only)

In the intrinsic evaluation experiment, both ORXESTRA and the Transformers vanilla were evaluated using the same dataset. This evaluation assesses their performance in a masked language modeling task, specifically in recovering the masked operation/operand. The *perplexity* values for each model are provided in Table 3.4. The best results in the table are highlighted in bold. A lower perplexity score indicates that the language model is better at predicting masked words.

The results clearly indicate that ORXESTRA outperforms the Transformers vanilla across all targets. This observation is not surprising, as Transformers XL, unlike Transformers vanilla, exhibits superior memory capabilities for handling long sequences. In contrast, Transformers vanilla is limited by the restricted number of tokens that can be

Target	M4	M7	A53	A72
Transformers vanilla	24.1	26.4	25.3	25.7
ORXESTRA	**19.2**	**23.1**	**22.2**	**21.8**

Table 3.4 – Perplexity scores obtained by ORXESTRA and the Transformers vanilla in the pretraining phase.

processed, which restricts its ability to capture dependencies beyond its specified context length. This hard sequence segmentation leads to context fragmentation, inefficient optimization, and, ultimately, a decline in performance.

3.3.2 Hyperparameters tuning

As often discussed in research papers using recurrent neural networks (e.g., [125, 93]), tuning hyperparameters is of utmost importance to guarantee the training convergence and the generalization to new data. However, this tuning phase is computationally expensive; thus, only a limited number of configurations can be tested. We show in the following how we have tuned (on validation data) our learning hyperparameters and how this can drastically improve the performance (in terms of MAPE) of ORXESTRA, Transformers vanilla, CATREEN and our closest competitor, ITHEMAL.

To reduce the hyperparameter exploration space, we did not tune the *architecture* of the models since we started with ITHEMAL (for LSTM) and PALMTREE (for Transformers) as the base architecture. We have studied the impact of three learning hyperparameters:

— The *optimization algorithm*, by comparing two optimizers: standard Stochastic Gradient Descent (SGD), used in ITHEMAL [129] and ADAM optimizer [98] widely used in deep learning.

— The *learning rate*. ITHEMAL uses an adaptable learning rate with an initial value of 10^{-1} which decreases by a factor of 1.2 every epoch after the first two epochs. We consider that 10^{-1} is a high learning rate, so we chose to explore lower values. However, a low learning rate considerably increases the training time. Thus, we have explored only three learning rates: two constant values 10^{-3} and 10^{-4}, and an adaptive value that starts from 10^{-2} and decreases at each epoch by a factor of 10 until 10^{-4}).

— The *loss function*, by comparing two regression losses: the one used in ITHEMAL (Mean Absolute Percentage Error) and the symmetric Mean Absolute Percentage Error loss function [38] which is neutral regarding under or over-forecasting and seemed more appropriate. Their formulas can be found on page 39 of this document.

The sensitivity of our models and competitors to hyperparameter tuning for the Cortex-M7 dataset is shown in Table 3.5.

Loss function	MAPE						sMAPE					
Optimizer	SGD			ADAM			SGD			ADAM		
Learning Rate	10^{-3}	10^{-4}	adapt	10^{-3}	10^{-4}	adapt	10^{-3}	10^{-4}	adapt	10^{-3}	10^{-4}	adapt
ITHEMAL	37.2%	25.6%	39.9%	18,2%	**15,2%**	17,1%	26.4%	19.8%	27%	19.1%	18.1%	18.8%
CATREEN	17.9%	16.9%	17.7%	15.8%	14.6%	14.9%	18.7%	16.9%	17.3%	15.7%	**14.3%**	15.1%
Transformers vanilla	17.8%	17.2%	17.6%	14.1%	**12.8%**	16.3%	17.0%	16.9%	17.5%	14.9%	14.9%	15.6%
ORXESTRA	16.8%	15.8%	16.9%	13.1%	**12.6%**	15.7%	17.2%	16.7%	17.6%	16.1%	14.0%	15.3%

Table 3.5 – MAPE performance of ITHELAM, CATREEN, Transformers vanilla, and ORXESTRA, for different learning hyperparameters (loss function, optimizer, learning rate) for Cortex-M7. The lower, the better.

Overall, the validation (training scores) results reported in Table 3.5 show that ORXESTRA generally outperforms the other models for both MAPE and sMAPE, the ADAM optimizer, with its adaptive learning rate adjustments, consistently yields lower error rates across models. A slower learning rate of 10^{-4} further ensures a more reliable convergence during training. ITHEMAL is consistently the poorest performer, and it is also shown that the hyperparameter selection, as done in the original version of ITHEMAL (MAPE, SGD, adaptative learning rate), is sub-optimal for our data, and we manage to improve it accuracy on validation data by $162,5\%$. We can conclude that the hyperparameter selection greatly impacts the model performance (i.e., the error estimated on the validation set). In the next sections, we use ORXESTRA, Transformers vanilla, CATREEN and ITHEMAL with their best-found hyperparameters as summarized in Table 3.6.

Hyperparameter	ITHEMAL	CATREEN	Transformers vanilla	ORXESTRA
Loss function	MAPE	sMAPE	MAPE	MAPE
Optimizer	"Adam"	"Adam"	"Adam"	"Adam"
Learning rate	10^{-4}	10^{-4}	10^{-4}	10^{-4}

Table 3.6 – loss function, optimizer, and learning rate used for ITHEMAL CATREEN, the Transformers vanilla, and ORXESTRA.

3.3.3 Prediction results on the test dataset

We generated two distinct test sets for extrinsic evaluation purposes. The first test set consists of 500 BBs with fewer than 50 instructions and 500 BBs with more than 50 instructions. The second test set (a subset of the first one) is created specifically for all the BBs whose timing can be successfully predicted by the Transformers vanilla. To accommodate the limitations of Transformers vanilla, which cannot handle BBs with token sequence sizes exceeding their capacity, we purposefully collected a second test set of BBs

with sizes smaller than those in the first test set. This adjustment was essential to ensure a fair comparison between the models.

In Table 3.7, we use the first test dataset, and thus we do not provide the results for the Transformers vanilla (which could only perform predictions on the second test dataset). The results include the Mean Absolute Percentage Error (MAPE), where lower percentages indicate better model performance. The best results in the table are highlighted in bold. Additionally, the Pearson correlation score is utilized as another evaluation metric to estimate how correlated the predictions are with the ground truth. In this case, higher scores indicate better model performance.

Target	M4		M7		A53		A72	
Scores	MAPE	Corr.	MAPE	Corr.	MAPE	Corr.	MAPE	Corr.
Neural Networks	26.4%	0.93	22.7%	0.92	38.4%	0.89	16.7%	0.98
ITHEMAL	14.4%	0.90	17.6%	0.90	10.1%	0.98	11.4%	0.98
CATREEN	8.8%	**0.99**	13.3%	0.96	8.5%	**0.99**	10.4%	0.98
ORXESTRA	**7.8%**	**0.99**	**9.6%**	**0.98**	**5.2%**	**0.99**	**6.9%**	**0.99**

Table 3.7 – Test results of Neural Networks (NN), ITHEMAL [130], CATREEN [8], and ORXESTRA on various ARM Cortex targets: M4, M7, A53, and A72. The results are based on the first test dataset, which includes a balance between the number of small and large-sized BBs. Evaluation metrics: mean absolute percentage error and Pearson correlation (Corr.).

Table 3.7 shows that ORXESTRA obtains better MAPE performance than all other techniques for all the target architectures. The second best-performing model is CATREEN, which, like ORXESTRA, considers the execution context of BBs. The worst-performing model is the Neural Network this shows the importance of accounting for the sequential information. The context-agnostic techniques, ITHEMAL and, as shown in Table 3.8 for the second test set, the Transformers vanilla are positioned after the context-aware techniques. The correlation is high for all models and better for the model designed to process sequential data.

The complexity of the target architecture plays a role in the final results, although its influence varies depending on the measurement method employed. When measuring timings on Cortex M4 and M7 using JTAG, we observe that errors on M7 are higher than for Cortex M4. This discrepancy is due to the deterministic nature of the Cortex M4 architecture, while M7 incorporates a cache with a random replacement policy, which introduces timing variability. However, this observation does not hold for more sophisticated architectures such as Cortex A53 and A72. For these processors, measurement methods involving software instrumentation were necessary, which introduced additional cycles into the measurements. The insertion of measurement instruments disrupts the execution, particularly affecting memory plans and cache behavior. As a result, the data (and

in particular, the timing labels) obtained for these processors are slightly less accurate compared to processors with a JTAG interface. Consequently, making a direct comparison between these architectures is challenging.

Table 3.8 reports the results obtained on the second dataset. We can notice that the trends observed in the previous table 3.7 remain consistent.

Target	M4	M7	A53	A72
Scores	MAPE	MAPE	MAPE	MAPE
ITHEMAL	10.0%	14.4%	12.1%	13.0%
CATREEN	9.6%	14.5%	10.3%	11.8%
Transformers vanilla	9.1%	13.8%	13.5%	13.3%
ORXESTRA	**8.7%**	**6.8%**	**6.1%**	**7.5%**

Table 3.8 – Test Results: Mean Absolute Percentage Error (MAPE) on Different Targets (M4, M7, A53, and A72) using the second test set. The Test Set is specifically chosen to be within the prediction capabilities of Transformers vanilla, ensuring a fairer comparison among models.

3.3.4 Impact of the context size

In the previous paragraph, we evaluated the importance of context awareness to make accurate timing predictions. However, an important question arises: how much context is necessary? To explore this, we conducted investigations on CATREEN and ORXESTRA for the different target architectures. Experimental results are reported in Table 3.9 and Table 3.10 respectively. The best results in the table are highlighted in bold.

Target	M4	M7	A53	A72
None	13.0%	26.1%	36.3%	18.2%
1	12.5%	15.2%	21.0%	18.4%
3	**8.8**%	15.5%	**8.5**%	**10.4**%
6	9.3%	**13.3**%	12.5%	15.5%
20	10.2%	14.2%	9.5%	11.4%

Table 3.9 – Impact of the context size (number of BB considered as context) on the Mean Absolute Percentage Error of CATREEN.

The results highlight the importance of context in timing estimates. Both CATREEN and ORXESTRA, when deprived of context, produce estimates akin to standard neural networks. As the context size grows, the prediction errors diminish. Yet, there's a limit to this improvement. For ORXESTRA, errors stabilize after including 3 BBs for the M4 architecture and 6 BBs for other processors. For CATREEN, most processors stabilize at 3 BBs, but M7 needs 6 BBs. This plateau in error reduction can be attributed to the

Target	M4	M7	A53	A72
None	12.5%	24.5%	34.6%	13.3%
1	11.9%	14.6%	20.5%	13.1%
3	**7.8**%	14.5%	22.9%	14.5%
6	8.8%	**9.6**%	**5.2**%	**6.9**%
20	9.2%	13.7%	8.3%	8.8%

Table 3.10 – Impact of the context size (number of BB considered as context) on the Mean Absolute Percentage Error of ORXESTRA.

inherent constraints of LSTM architectures. Overly long contexts can overload the context vector, making it less effective within the set hyperparameters of both models.

3.3.5 Impact of the basic block size

Figure 3.5 displays the prediction errors (in mean absolute error [5]) for the six sets of 500 BBs in the test dataset: 500 BBs with less than 10 instructions (in the left), 500 BBs with a number of instructions between 10 and 20, 500 BBs with a number of instructions between 20 and 30, 500 BBs with a number of instructions between 30 and 40, 500 BBs with a number of instructions between 40 and 50, 500 BBs with a number of instructions between 50 and 100, and 500 BBs with more than 100 instructions (in the right). We observe that ORXESTRA exhibits a lower error trend across BBs of different sizes for all architectures. CATREEN demonstrates low error rates when predicting larger-sized BBs than smaller-sized ones. On the other hand, ITHEMAL exhibits a consistent trend, as we observe that there is a significant increase in error when predicting long BBs (especially those with more than 100 instructions). In terms of ORXESTRA errors, it is worth noting that average errors tend to be lower for the more complex architectures, such as Cortex A53 and A72. Conversely, processor architectures that strictly adhere to an in-order pipeline can result in high errors for ORXESTRA. To understand the factors contributing to smaller errors in the more complex architectures, we examined the standard deviation of execution times for basic blocks (BBs) in the initial test set. The obtained results for each architecture were as follows: M4 - approximately 252 cycles (due to the absence of a cache), M7 - approximately 113 cycles, A53 - approximately 83 cycles, and A72 - approximately 64 cycles. This indicates that the reduction in error can be attributed to the inherent nature of the processor. Superscalar and out-of-order architectures have the ability to rearrange the order of instruction execution, thereby mitigating delays caused by instruction dependencies and resulting in less variability in execution times. The variability in estimates significantly impacts the predictive capabilities of ORXESTRA.

5. Mean Absolute Error: MAE $= \frac{1}{n} * \sum_{1}^{i=n} |prediction_i - truth_i|$

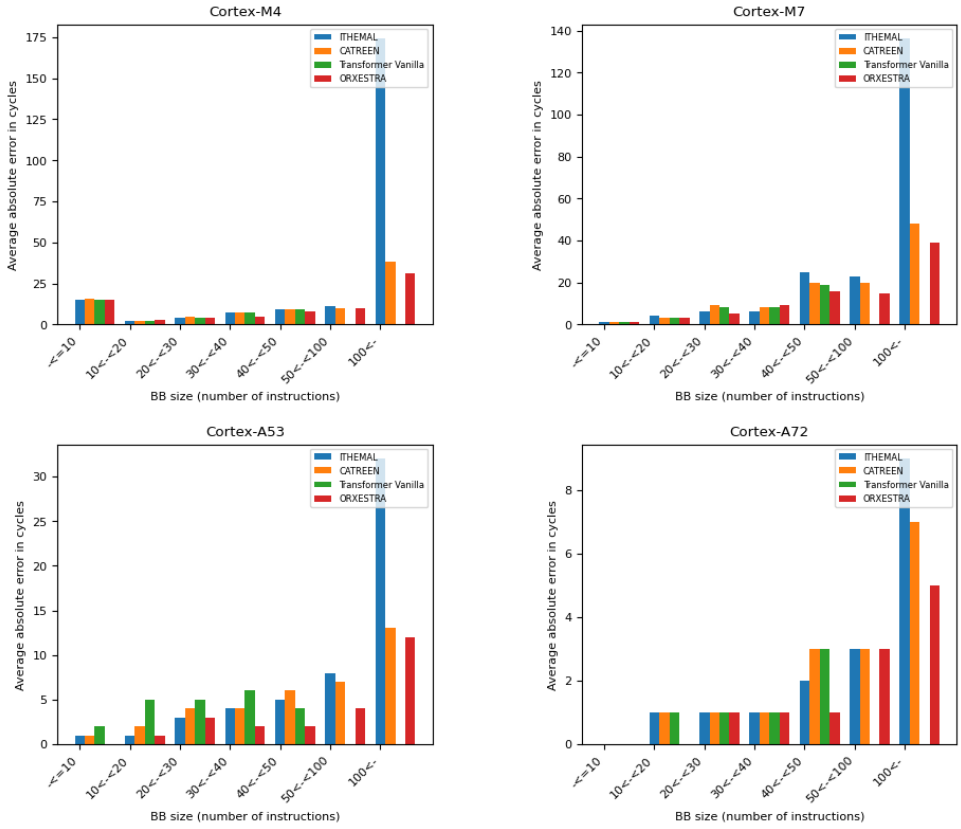

Figure 3.5 – Mean absolute cycle error (average number of cycle error) boxplot comparison of ITHEMAL (blue), CATREEN (orange), Transformers vanilla (green), and ORXESTRA (red) for different processors (M4, M7, A53, A72) and six Categories of basic blocks. The most left category represents basic blocks with a size of 10 or less instructions (-<=10), while the most right category includes basic blocks with a number of instructions exceeding 100 instructions (100<). Each subfigure represents a processor.

3.3.6 Optimization effect on prediction

In this experiment, we want to use different GCC optimization levels (O0, O1, O2, O3) to compile and generate the test dataset. This allows for an investigation into how sensitive the models are to changes in optimization levels. The goal is to answer the question: are the models robust enough to maintain performance across different optimization levels? Table 3.11 gives the results of this experimentation, and we observe that:

— ORXESTRA: Consistently outperforms the other models across all targets and optimization levels.

— It is evident that Neural Networks exhibit the highest error in results, making them the most sensitive to compiler optimization. ITHEMAL consistently outperforms Neural Networks and occasionally surpasses CATREEN. ITHEMAL shows high variability in MAPE scores, contrary to CATREEN and ORXESTRA, making

ITHEMAL not only influenced by the model hyperparameters setting as discussed in Section 3.3.2 but also by the optimization level used to compile the testing data.

— Optimization Levels: The optimization levels do not seem to consistently impact error rates across different models and targets, which is interesting. Additionally, it is worth noting that the error rates for each model do not vary significantly when changing the optimization levels, indicating a level of robustness in the models' performances across different optimization settings.

Target	M4				M7				A53				A72			
Optimization	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3
Neural Networks	28.1%	23.2%	21.9%	26.4%	21.1%	19.9%	28.3%	22.7%	25.2%	40.4%	37.1%	38.4%	18.2%	17.1%	16.4%	16.7%
ITHEMAL-tuned	14.1%	14.5%	14.5%	14.4%	18.2%	18.2%	17.7%	17.6%	9.1%	11.0%	10.6%	10.1%	12.2%	12.3%	11.5%	11.4%
\CATREEN	8.9%	8.2%	8.9%	8.8%	13.4%	13.1%	12.8%	13.3%	9.7%	9.7%	9.2%	8.5%	11.1%	11.4%	10.8%	10.4%
\ORXESTRA	**7.6%**	**7.7%**	**7.7%**	**7.8%**	**8.9%**	**8.2%**	**9.9%**	**9.6%**	**6.2%**	**6.1%**	**6.3%**	**5.2%**	**7.9%**	**7.2%**	**7.4%**	**6.9%**

Table 3.11 – MAPE performance of ORXESTRA, CATREEN, ITHEMAL and Neural Networks across various GCC optimization levels (O0, O1, O2 and O3) and architectural targets

3.3.7 Inference throughput

Table 3.12 displays the instruction rate per second achieved by each machine learning model. Interestingly, this time is found to be independent of the complexity of the target processor architecture, so we report the average over all processors. The throughput calculation is based on the 1000 basic blocks utilized in the previous experiments (the first test set). To ensure a fair comparison, we present the results in the first column with a batch size of 1, followed by the results with a batch size of 32 in the second column for all techniques. Notably, neural networks demonstrate the highest execution speed, despite their lower accuracy. Transformers Vanilla, which does not consider the execution context, follows closely. ORXESTRA, which processes this context, provides a better execution speed compared to LSTM-based networks (ITHEMAL and CATREEN), which require sequential processing of each instruction. Consequently, CATREEN is the slowest among them due to the additional context processing involved.

	Throughput Instruction/second	
Batch size	1	32
Neural Networks	5131	162140
ITHEMAL	1627	45379
CATREEN	1356	32644
Transformers vanilla	3809	112468
ORXESTRA	2691	74172

Table 3.12 – The mean throughput over all processors, when treating 1000 BB for each technique (with a batch size of 1 and batch size of 32).

3.4 Conclusion

In this chapter, we delineated the capabilities of CATREEN and ORXESTRA, two distinct machine learning-based timing predictors. CATREEN leverages Long Short-Term Memory (LSTM) networks to estimate the average execution time of a program's basic blocks, taking into account the *execution context* informed by the sequence of previously executed basic blocks. In contrast, ORXESTRA employs the Transformers XL architecture for timing prediction, while also considering the execution context.

The experimental findings suggest comparative advantages, particularly favoring ORXESTRA, whose timing accuracy predictions are 28% superior to the context-aware CATREEN and are executed 98% faster. However, despite the promising results, there are several areas of potential improvement in this study that warrant further investigation. Below, we outline the key limitations and suggest areas for future enhancement:

— **Context size limitations:** Performance degradation when dealing with a large number of basic blocks is a concern. Further research is needed to ascertain whether this limitation is attributed to the dimensions of our Transformers XL and LSTMs network. If a larger network size can deliver improved accuracy, then leveraging parallel GPU-based training would become crucial.

— **Context representation:** Currently, the context is represented as a sequence of basic blocks (BBs). Other, more efficient representations should be explored, notably graph neural networks, which could process a control flow graph and ensure a more accurate representation of a BB's context.

— **Random cache replacement policy:** The unpredictable nature of a random cache replacement policy poses challenges for accurate execution time prediction. A possible countermeasure could be the integration of performance counters to track cache activities. By selecting a representative set of executions based on distinct cache miss ranges, we could more accurately capture the average memory access patterns, thereby refining our estimates related to execution time variability.

By systematically addressing these identified limits and incorporating the aforementioned solutions, the performance and practical applicability of ORXESTRA and CATREEN in real world computational settings are expected to be substantially enhanced.

Chapter 4

Towards Refined WCET Estimation: The Potential of Transformers XL

In the previous chapter, we introduced two novel machine learning techniques, ORXESTRA and CATREEN, designed to accurately account for instruction-level dependencies both within a basic block and in relation to previously executed basic block sequences. This *context* is a crucial input for our context-aware models.

In this chapter, we propose CAWET, a novel Hybrid WCET Technique that employs Machine Learning (HT-ML). CAWET is an acronym for **C**ontext-**A**ware **W**orst-case execution time **E**stimation using **T**ransformers. It leverages the Transformers XL advanced machine learning algorithm [44], similar to ORXESTRA from Chapter 3. Since ORXESTRA has shown superior performance over CATREEN in both accuracy and execution speed, we have adapted it to predict the worst-case execution time of basic blocks. Unlike other HT-ML methods that focus solely on 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 natural language, where the timing of a BB depends not only on its instructions sequence but also on the sequence of BBs executed before it.

Hypotheses

This section presents the assumptions guiding the functioning of our proposed technique, i.e., CAWET.

— In this chapter, our primary focus is on the estimation phase; our objective is to determine the worst possible context, i.e., the sequence of previously executed basic blocks leading to the worst-case execution time of the target basic block. In the training phase, we only retrain the Transformers XL to predict the WCET estimation.

— Similar to Chapter 3, we treat the size of the execution context as a hyperparameter. We limit our analysis to a predetermined number of basic blocks, as excessive numbers can result in the path explosion problem during the search for all possible

contexts of the target basic block.

— CAWET can estimate the WCET for basic blocks in complex architectures with limited documentation. It can capture pipeline and some cache effects during the estimation of the basic block's WCET, making it also ideal (as WE-HML) for aeronautics applications in DAL B and C categories [20].

The rest of this chapter is organized as follows. Section 4.1 presents the CAWET HT-ML technique. The experimental methodology for evaluating it is detailed in Section 4.2, and experimental results are given in Section 4.3. We conclude in Section 4.4. The main content of this chapter has been published at ECRTS 2023:

"**Abderaouf N., AMALOU**, Elisa FROMONT, and Isabelle PUAUT. "CAWET: Context-Aware Worst-Case Execution Time Estimation Using Transformers." The 35th Euromicro Conference on Real-Time Systems, (ECRTS 2023). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023."

4.1 The CAWET approach

CAWET consists of two main stages: *training* and *deployment* (or *estimation*). As in all systems using Transformers, the Transformers model is first pretrained in the learning phase to comprehend the vocabulary (in our context, assembly language). Then, the model is finetuned using extensive measurements on various basic blocks extracted from real codes. In this finetuning 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 due to consideration of the execution context of all basic blocks. CAWET is evaluated on processors of varied complexity, including the cortex-M4, the more advanced cortex-M7 that features a cache, and the even more sophisticated cortex-A53. The quality of the WCET estimates produced by CAWET is compared to those produced by WE-HML, the HT-ML technique closest to CAWET [7], on 13 programs from the TACLeBench benchmark suite [64]. Our results show that CAWET produces better estimates than its competitors on more diverse architectures.

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 4.1.1.

The two main phases of CAWET, training (using Transformers XL) and prediction (i.e., deployment), are then respectively presented in Sections 4.1.2 and 4.1.3.

4.1.1 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 4.1.

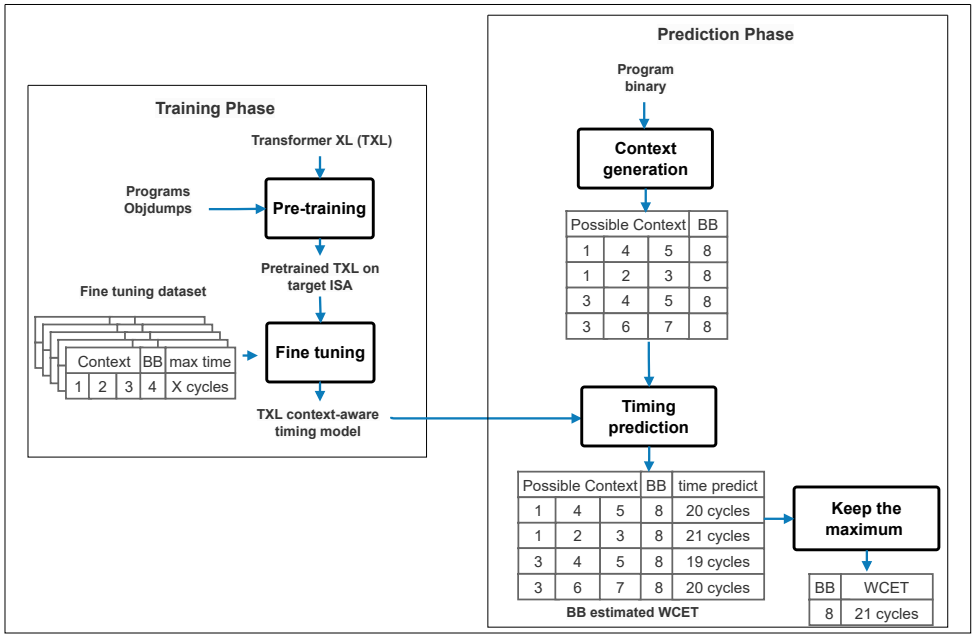

Figure 4.1 – Overview of CAWET

In the training phase (left block of Figure 4.1), the Transformers 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 [55]. Then, the model is finetuned using extensive measurements on a large set of BBs extracted from real code. In this finetuning 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 4.1), the WCET of each BB is determined. Since there might be different execution paths leading to the BB under study, the prediction operates on the set of contexts corresponding to these paths, with care taken to avoid combinatorial explosion, as further explained in Section 4.1.3. The prediction phase

first computes the list of contexts of the BB under study (BB number 8 in the Figure). The result 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.

4.1.2 Training phase using Transformers XL

Transformers are neural network architectures 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 [165], 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 [165], the original Transformers 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) [44] were introduced. A TXL is a variation of the Transformers 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 Transformers 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).

The training of a TXL consists of two stages (*pretraining* and *finetuning*). During the pretraining 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 [55] 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 pretraining phase, thousands of disassembled binary programs are used without needing labeled information. Details about the hyperparameters of the TXL architecture are provided on the Page 96.

In the finetuning 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* [103], a well-known tokenization technique trained in our work on the target assembly instructions. The training dataset

for the finetuning 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.

4.1.3 Prediction phase

CAWET predicts the WCET of BBs by considering their different execution contexts and retaining the largest one. The results from CAWET can then be integrated into a static WCET estimation tool. In this section, we first introduce the foundational concepts and notations upon which CAWET is built. Following that, we delve into the specifics of context generation. We then outline how the WCET of a BB is derived from the predictions, culminating in the final calculation of the program's overall WCET.

Concepts and notations

The concepts and notations used in CAWET are standard concepts used in compilers. They are illustrated in Figure 4.2, which will be reused later to illustrate how CAWET works.

Definition 3 (SESE regions, SESE trees.) *A Single Entry Single Exit (SESE) region, as defined in [94], 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, and constructed in linear time [94].*

An example of CFG (with 7 BBs numbered from 1 to 7), and its SESE regions is depicted in Figure 4.2 (A). The dotted arrow in the figure represents the back edge of the loop whose body is composed of BB 5 and 6. The SESE tree that corresponds to the CFG is depicted in Figure 4.2 (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 4 (Cyclomatic complexity.) *Cyclomatic complexity is a software metric that measures the number of independent paths through a program or a CFG [61]. 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 4.2 (B).

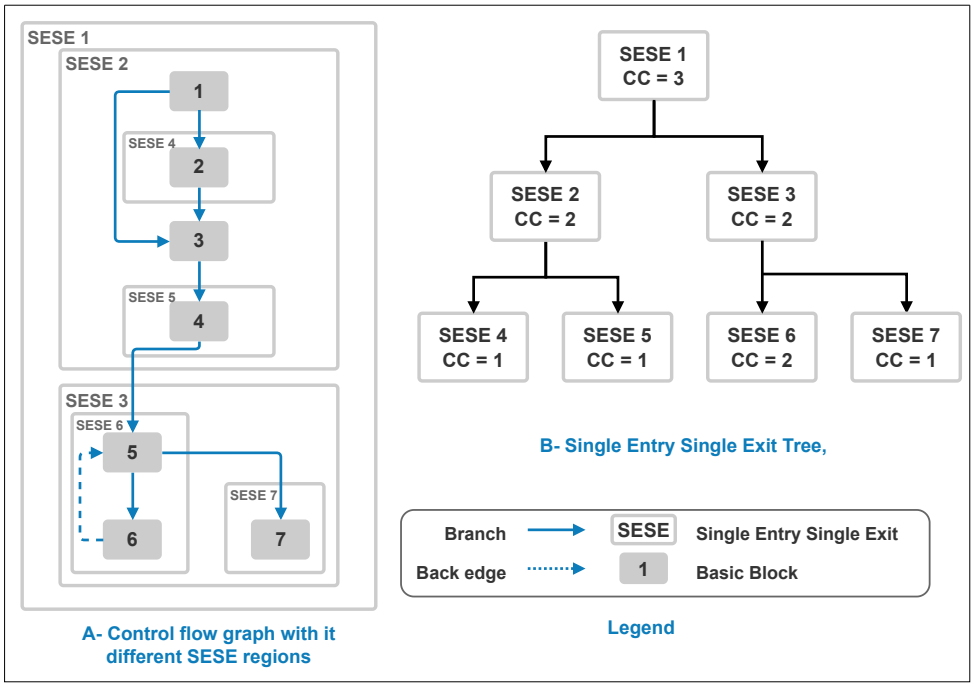

Figure 4.2 – A CFG example transformed into a SESE tree and annotated with cyclomatic complexity.

Context generation

For each basic block, we aim to identify the potential preceding sequences of basic blocks that could be executed. However, determining all possible paths in a graph can be computationally intensive. To mitigate this challenge, we employ a divide-and-conquer approach leveraging the program's SESE tree. In the example of Figure 4.2, 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 [160] (DFS) to enumerate all possible paths [1]. 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 4.3 step 1 using the CFG and SESE in Figure 4.2

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

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 4.3).

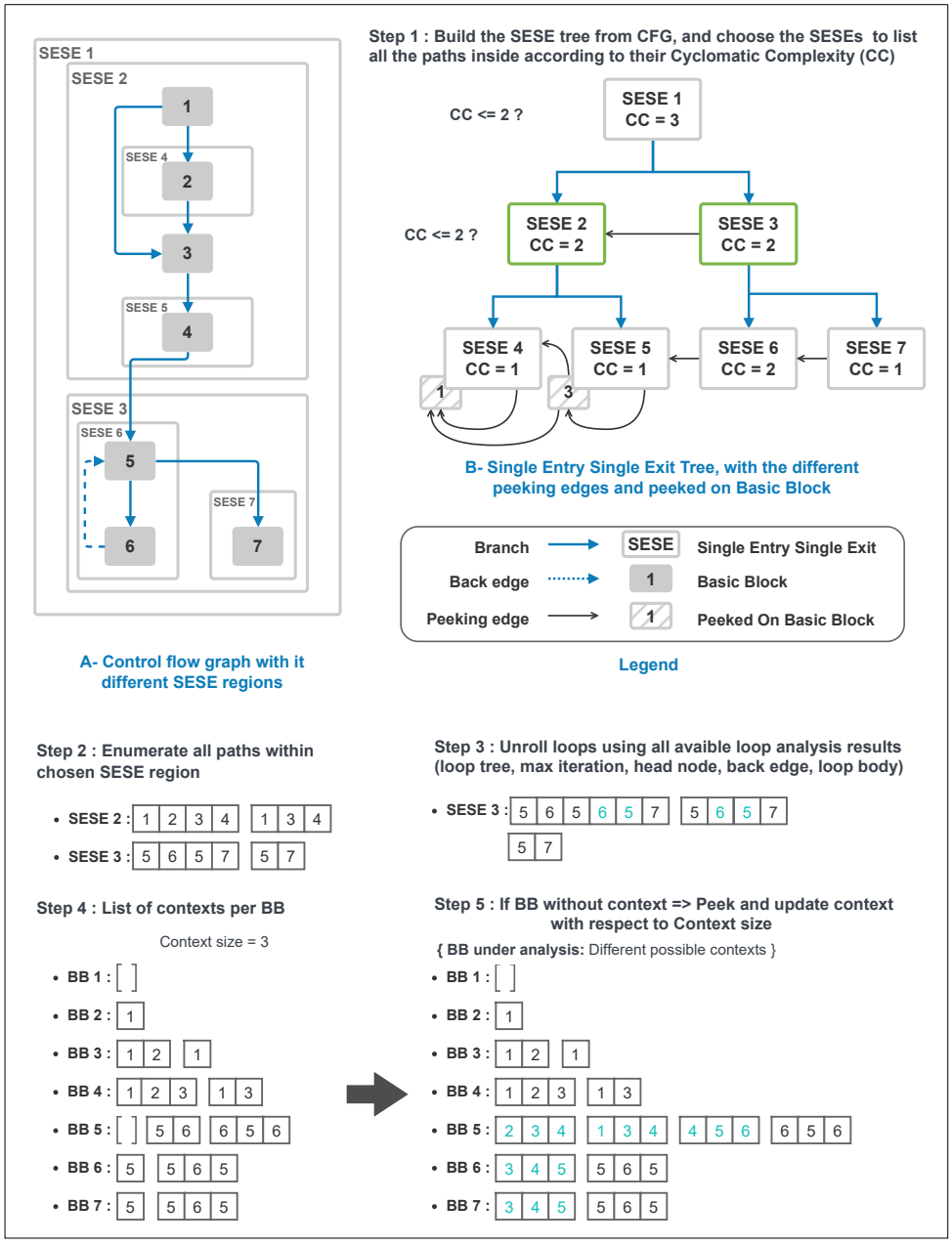

Figure 4.3 – Example of the different steps for context generation, where the cyclomatic complexity limit is set to 2 and the context size is set to 3 BBs.

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 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 [2].

The result of the loop unrolling process on our example is given in Figure 4.3 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 4.3 step 4. The size of each context is limited to the *context size* hyperparameter of CAWET.

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 the SESE tree from Figure 4.3; 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 4.3 step 5 shows that the context of BB 5 can be augmented by peeking at the execution trace of SESE 2.

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 typically signifies the initial execution of the basic block within a loop, which tends to be longer. In contrast, other values might indicate subsequent, potentially shorter, execu-

2. 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.

tions of the same block. Due to a limited context size, predicted timing values might be overly optimistic. To address this, we explore a technique in later sections that incorporates static cache analysis, adding its overhead to CAWET's timing values. Subsequently, the WCET of BBs is integrated into a static WCET estimation tool, using established methods like IPET [172], to determine the program's overall WCET.

4.2 Experimental setup

In this section, we detail the experimental setup employed to assess CAWET. The datasets for pretraining and finetuning are provided in Section 4.2.1 and they are consistent with the one utilized in Chapter 3. For finetuning, while the dataset remain the same, the labels employed in this chapter represent the maximum observed execution time of the basic block. Meanwhile, the testing dataset, derived from TacleBench, aligns with the one referenced in Chapter 2. Section 4.2.2 introduces the context-agnostic baselines against which CAWET's performance is benchmarked. Subsequent sections, namely Section 4.2.3, Section 4.2.4, and 4.2.5, elucidate the software/hardware environments and the configurations for both the learning and prediction phases of CAWET.

4.2.1 Dataset and benchmarks

For CAWET, which uses Transformers XL [43], pretraining and finetuning are essential steps. The information in this subsection is redundant with that of Chapter 3 (training ORXESTRA) and Chapter 2 (testing on Taclebench [64] programs), but for a clear understanding of this chapter, we decided to duplicate them while adding the information specific to CAWET.

We have pretrained CAWET on a large number of BBs in order for the Transformers to learn the assembly language under study, using CodeNet [140]. 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 pretraining set is also used to build a vocabulary model with *sentence piece* [103]. 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 finetune CAWET on basic blocks with their context, we have used a diverse and publicly available set of programs: *The Algorithms*[3], *MiBench* [75] and

3. Available here: https://github.com/TheAlgorithms/C

Polybench [174]. Table 4.1 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 4.1 – 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 4.2 – 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 [64] whose characteristics are given in Table 4.2. 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 [7], 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.

4.2.2 Context-agnostic baselines

CAWET is evaluated by comparing it to two context-agnostic WCET predictors. The first is a Multi-Layer Perceptron regressor, loosely referred to as a Neural Network (NN), which we train under the same conditions as the finetuning step of CAWET (same dataset,

same loss function). 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 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 [7], a hybrid ML-based WCET estimation technique presented in Chapter 2. One of the best-performing ML algorithms of [7] (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.

4.2.3 Hardware and software setups

For training and validating CAWET, accurate timing values are essential. The methodology to gather these values must avoid any interference with the program's execution, also known as the *probe effect*. In this process, we remind how we set up the measurement process at each Chapter of this document:

Chapter 2. We measure synthetically generated C code, often referred to as basic blocks, using software instrumentation. However, these measurements are subject to operating system interferences, which necessitated the use of a probabilistic method to filter the unwanted noise for each basic block measurement.

Chapter 3. Whenever available on the target processor, we utilize a hardware-based measurement solution (JTAG) and operate on bare metal. This ensures precise measurements for each basic block, eliminating the noise that could be introduced by the operating system. If software instrumentation is employed, we insert measurement code both before and after the completion of a basic block. By leveraging annotations on maximum loop iterations combined with GDB (the GNU Debugger), we can match each execution time to its corresponding basic block. Both JTAG and GDB are also used to retrieve the execution context.

In this Chapter. We build upon the measurement protocol from Chapter 3, extracting 1,000 measurements for each basic block. We observe that the variability is less pronounced compared to one encountered in Chapter 2. This is because we either use JTAG, which avoids the probe effect issues, or we employ more refined instrumentation

where we measure instruction blocks that are smaller than those synthetically generated in Chapter 2, thereby avoiding noises. In this case, we use the maximum observed execution time for each basic block (the maximum from 1,000 of measurements).

Our experiments encompass various Arm processors, the features of which are summarized in Table 4.3. It includs the Cortex-M4, Cortex-M7, and Cortex-A53. We focus initially on the Cortex-M4 for its deterministic nature before moving to the more advanced Cortex-M7 and then the Cortex-A53. Notably, in contrast to Chapter 3, we exclude the out-of-order Cortex-A72 processor from our study. The definition of context in such processors differs, as it can encompass both the preceding and succeeding basic blocks due to their ability to buffer and reorder instructions.

Table 4.3 – 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 [7])	Software	Linux	In-order/8	Yes	Yes data and instruction cache, L2, random replacement policy

4.2.4 Setup for the learning phase

PyTorch was used to implement the learning models, which were then trained on a Tesla V100 GPU. 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 4.1, which tends to penalize underestimations more heavily than overestimations. The main reason why we did not do it in Chapter 2 is because we used Scikit-learn [139], which offers a variety of regression algorithms, but it does not provide the flexibility to customize the loss function (as the loss function is hardcoded in the library). We also incorporated an additional penalty **during training** for predictions that underestimated the execution time, according to Equation 4.2. 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 $(target - prediction)$ [4].

$$RMSLE(target, predict) = \sqrt{(\log(target + 1) - \log(predict + 1))^2} \tag{4.1}$$

$$UsedTarget = \begin{cases} target & if \quad target \leq prediction \\ target + (target - prediction) & if \quad target > prediction \end{cases} \tag{4.2}$$

4. The goal is to make our machine learning model minimize underestimation of the predicted WCET.

4.2.5 Setup for the prediction phase

The CFG, the SESE tree, and the loop tree are generated by the Heptane WCET estimation tool [79]. 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 in Table 4.2). 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 of the target architecture.

4.3 Experimental results

The quality of WCET predictions for the Cortex M4 and Cortex M7 architectures is evaluated in Sections 4.3.1 and 4.3.2. The effect of the different features of CAWET on the quality of the predictions is studied in Section 4.3.3. Finally, CAWET is evaluated in Subsection 4.3.4 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 [7] from Chapter 2.

4.3.1 Quality of WCET predictions for the Cortex M4

Table 4.4 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 4.2.2. 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 [5] 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.

Impact of the context size. Table 4.5 shows the considered context size's impact on the prediction quality. Four values are considered: 0 (no context), 1 BB as context, 3

5. Mean Absolute Error: MAE $= \frac{1}{n} * \sum_{1}^{i=n} |Error_i|$

Table 4.4 – 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**

BBs as context, and 20 BBs as context.

Table 4.5 – 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.

4.3.2 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 4.6 evaluates WCET predictions produced by CAWET and the baseline NN for the Cortex M7, using a context size of 6 for CAWET.

Table 4.6 – 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 4.7.

Table 4.7 – 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**

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.

4.3.3 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 unrolling (C), and finally, applying cache analysis (D). The results in Table 4.8 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 4.8 – 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

Table 4.9 – 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

4.3.4 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 [7], 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 [143], where we set the probability to 10^{-3} to provide another reference point than the

MOET.

Table 4.9 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.

4.4 Conclusion

In this chapter, we introduced CAWET: a hybrid approach that estimates the worst-case execution time of individual basic blocks within a program. Our approach uses static techniques to identify the longest execution path and Transformers XL [43] to predict the WCET of each basic block. By incorporating the execution context of preceding basic blocks, CAWET effectively captures the complexity of the processor's microarchitecture pipeline and partial cache effects, eliminating the need for explicit modeling of the target processor. Empirical evaluations conducted on the TacleBench benchmarks for various processors revealed that CAWET consistently avoids underestimating execution times, presenting a less pessimistic outlook compared to its competitors. While challenges remain, such as the need for extended context to enhance prediction accuracy and reduce pessimism (ensuring comprehensive cache effects and branch predictor considerations), CAWET offers a promising solution for predicting worst-case execution times for complex processors that are not well-documented.

Chapter 5

Conclusion and future works

This document introduces novel methods for estimating worst-case execution time (WCET) and average-case execution time (ACET) in complex computer architectures with limited documentation. The key contributions of this thesis are summarized in Section 5.1, while Section 5.2 outlines prospective research directions.

5.1 Key contributions

Below is a concise summary of our key contributions:

Hybrid methodology for enhanced WCET estimation (WE-HML). As a preliminary work during this thesis, we introduced WE-HML [7], a method designed to estimate WCET on modern processors where detailed knowledge of their inner workings is not always available. WE-HML is a hybrid technique: it uses WCET static methods to estimate the longest path, while basic machine learning techniques help to estimate the WCET of individual basic blocks. A unique feature of WE-HML is that it works directly on binary code, offering more precise learning than methods that rely on source or intermediate code. It is trained on a vast collection of automatically generated programs, ensuring quality learning. Additionally, WE-HML has a special way of accounting for data caches; training data considers the worst effects of caches, which is crucial for accurate WCET estimation. However, despite its initial promise, the approach tends to overestimate execution times. A contributing factor to these pessimistic estimations was the machine learning model's reliance on static features, which failed to account for the dependencies between instruction sequences within individual basic blocks.

Incorporation of Natural Language Processing (NLP) techniques for ACET estimation. As a pivot to refine our techniques, we transitioned our focus from worst-case scenarios to average-case execution times. The motivation behind this shift is to simplify the integration of NLP techniques for capturing intricate dependencies between instruction sequences. We tested various deep learning architectures such as LSTM [82], BERT [54], and Transformers XL [43] to evaluate their capability for contextualizing basic block execution times. Among these, Transformers XL distinguished themselves as the most accurate model for the task.

Context-aware WCET estimation using Transformers (CAWET). Building upon the successful integration of NLP techniques for average-case scenarios, a pressing question emerged: "How can one identify the *worst-case context* for a given basic block and then employ Transformers XL to estimate its WCET?" Addressing this inquiry, our third contribution—designated as CAWET—introduces a new mechanism to automatically identify every feasible execution context leading into a specific basic block. CAWET enhances the accuracy of WCET estimation by considering all possible short contexts. Taking advantage of the capabilities of Transformers XL, we integrated these findings into a static WCET analysis tool to create another hybrid methodology for estimating WCET. This enhanced tool effectively mitigates the overly pessimistic estimations of our initial WE-HML model, thereby increasing our approach's overall accuracy and applicability.

5.2 Open issues and future perspectives

We have presented various methods for estimating both worst-case and average-case execution times. As we look ahead, we provide short-term and medium-term recommendations to further improve these approaches.

Future works in the short-term

Toward explainable models. Machine learning models such as neural networks offer powerful capabilities for estimating WCET and ACET. However, their inherent "black-box" nature poses the challenge of understanding the decision-making process. This lack of transparency can be a roadblock when these models are employed in crucial tasks such as WCET estimation, where understanding the rationale behind predictions is essential for identifying errors or anomalies in our models. This is where the concept of "explainability" gains importance [134]. Explainability in machine learning is like a clear window into how a model thinks. It helps us understand why a model makes a certain decision. This not only builds trust in the model's results but also lets the user spot any problems or biases in how it is working. As a first attempt, with a colleague from the NOP CominLab project [1] (Safe and Efficient Intermittent Computing for a Battery-less IoT), we proposed a new ecosystem called WORTEX: WORst-case execution Time and Energy consumption estimation using eXplainable machine learning. Our proposed machine learning-based technique aims to improve timing and energy model interoperability. Figure 5.1 provides an illustrative example in which we employ the SHAP [121] (SHapley Additive exPlanations) technique to quantify the contributions of each static feature to the execution time for neural networks timing model. Specifically, this visual-

ization highlights how the presence of memory-access instructions (values different from 0) significantly influences the execution time. Conversely, their absence (values equal to 0) leads to a reduction in execution time. This behavior aligns well with the operational characteristics of the MSP430 processor, where any memory access operation is intrinsically more time-consuming compared to instructions that do not require memory access. However, explaining the decisions of more complex models like LSTM (CATREEN) and Transformers XL (ORXESTRA), especially when applied to the timing estimation of processors with limited documentation, presents new challenges. A significant limitation is the absence of a clear ground truth for the interpretations, given that many processors are protected by intellectual property rights. This limitation underscores the potential value of porting our solution to open-source instruction sets, such as RISC-V. By doing so, we can validate our explanations in a more transparent environment, setting a precedent for future works in this domain.

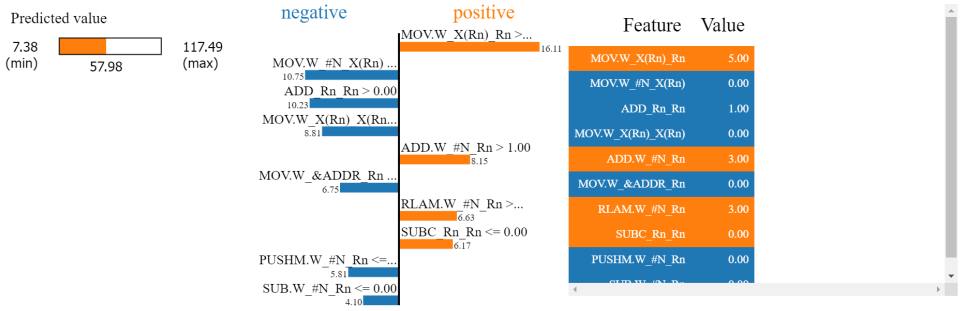

Figure 5.1 – Plot showing feature impacts on timing prediction for a basic block on MSP430.

Future works in the medium-term

Increasing context size. In Chapter 3, we have noticed a trend: as the context size increases (referring to the number of basic blocks executed immediately before the target basic block whose execution time is being estimated), there comes a point where the error score also starts to rise. Even though we have pushed our available GPU to its memory limits and utilized the most extensive possible network configurations for both LSTM and Transformers, we believe that this setup might not be adequate for learning a large context. Moreover, this could lead to the loss of important dependencies and information, including cache effects and branch predictor history. As a starting point for solving this limitation, we recommend delving into a larger set of learnable parameters (larger LSTMs and Transformers), potentially by using multiple GPUs. This approach could help us consider more basic blocks, thereby offering a more precise understanding of cache effects and branch predictors.

However, it is worth noting that an execution trace (sequence of basic blocks) can become excessively large, potentially exceeding the processing capacity of even the larger LSTM or Transformer models. It might be more meaningful if the context, formed by the preceding "N" basic blocks, is substituted with the control flow graph that contains the basic block whose execution time we aim to estimate. A promising direction is the use of graph-based machine learning architectures [173]. These architectures should naturally treat the Control Flow Graph (CFG) of the program in question alongside the basic block under analysis. The question that we leave open is how to integrate both the CFG and the target basic block into our prediction models to be able to capture a more comprehensive view of the execution context and potentially lead solutions like CATREEN and ORXESTRA to more accurate ACET and WCET predictions.

These proposals remain hypothetical, and nothing can be guaranteed until we evaluate whether our models can effectively accommodate more context.

Machine learning for timing estimation in multicores. Existing works [26, 41] have utilized machine learning to predict contention [1] in multicore processors. In our case, predicting the WCET in such systems already contradicts the hypothesis behind the WCET estimation of a task, which is supposed to represent the worst execution time of a task in isolation. However, it is more realistic to consider the nature of multicore processors, especially since many of today's processors, even embedded ones, are multicores. A future direction to extend this work to multicore systems lies in extending the notion of "context awareness" to account for activities that execute concurrently on other cores. Importantly, how can this be addressed while working with processors that have limited documentation?

A prospective approach might involve building on methodologies from prior works like WE-HML. Here, machine learning could be trained to understand various factors, such as the influence of other cores and cache pollution. Techniques such as [35] could be employed to simulate these conditions during the training phase. For predicting the timing, tools such as StAMP [50] might be considered to profile interference to estimate the *inter-core interference factor*.

However, some questions are still open:

Cache pollution in multicores. How can the cache pollution, as done in the WE-HML approach, be effectively adapted to multicore systems? How can we distinguish between pollution coming from a loop nest and that caused by interference from other cores?

Interference artificial simulation. When simulating inter-core interference by

1. Contention refers to the measurable delay or performance degradation caused by multiple cores competing for shared resources.

initiating parallel memory accesses on other cores, how can we ensure that this simulation accurately represents real-world scenarios?

Training challenges. Finetuning Transformer XL took time. Is there a more efficient solution to train the Transformers XL without needing to duplicate them for each pollution factor and for each inter-core interference factor?

BIBLIOGRAPHY

[1] URL: https://project.inria.fr/nopcl/.

[2] Andreas Abel and Jan Reineke, « nanoBench: A low-overhead tool for running microbenchmarks on x86 systems », *in*: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2020, pp. 34–46.

[3] Andreas Abel and Jan Reineke, « uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures », *in*: *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–14.

[4] Yuan Ai, Mugen Peng, and Kecheng Zhang, « Edge computing technologies for Internet of Things: a primer », *in*: *Digital Communications and Networks* 4.*2* (2018), pp. 77–86.

[5] Peter Altenbernd et al., « Early execution time-estimation through automatically generated timing models », *in*: *Real-Time Systems* 52.*6* (2016), pp. 731–760.

[6] Abderaouf N Amalou, Elisa Fromont, and Isabelle Puaut, « CAWET: Context-Aware Worst-Case Execution Time Estimation Using Transformers », *in*: *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[7] 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)*, IEEE, 2021, pp. 31–40.

[8] 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.

[9] Marcos Amaris et al., « A comparison of GPU execution time prediction using machine learning and analytical modeling », *in*: *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, IEEE, 2016, pp. 326–333.

[10] Marcos Amaris et al., « A simple BSP-based model to predict execution time in GPU applications », *in*: *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, IEEE, 2015, pp. 285–294.

[11] Bjorn Andersson et al., *Assessing the Use of Machine Learning to Find the Worst-Case Execution Time of Avionics Software*, tech. rep., United States. Department of Transportation. Federal Aviation Administration . . ., 2023.

[12] llvm-dev Andrea Di Biagio, *[llvm-dev] [RFC] llvm-mca: a static performance analysis tool*, Mar. 2018, URL: `https://lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html` (visited on 12/07/2021).

[13] Todd Austin, Eric Larson, and Dan Ernst, « SimpleScalar: An infrastructure for computer system modeling », *in*: *Computer* 35.*2* (2002), pp. 59–67.

[14] Philip Axer et al., « Building timing predictable embedded systems », *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 13.*4* (2014), pp. 1–37.

[15] Clément Ballabriga et al., « 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*, Springer, 2010, pp. 35–46.

[16] Iain Bate, David Griffin, and Benjamin Lesage, « Establishing Confidence and Understanding Uncertainty in Real-Time Systems », *in*: *ACM International Conference on Real-Time Networks and Systems*, 2020, pp. 67–77, ISBN: 9781450375931.

[17] Adam Betts and Guillem 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.

[18] Jan Bielecki, « Estimation of execution time for computing tasks », *in*: *Cluster Computing* (2022), pp. 1–14.

[19] Nathan Binkert et al., « The gem5 simulator », *in*: *ACM SIGARCH computer architecture news* 39.*2* (2011), pp. 1–7.

[20] Jean-Paul Blanquart et al., « Criticality categories across safety standards in different domains », *in*: *Embedded Real Time Software and Systems (ERTS)*, 2012.

[21] Armelle Bonenfant et al., « Early WCET prediction using machine learning », *in*: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, OASICs, Dagstuhl Publishing, 2017, pp. 5–1.

[22] Armelle Bonenfant et al., « Early WCET prediction using machine learning », *in*: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, OASICs, Dagstuhl Publishing, 2017, pp. 5–1.

[23] Frédéric Boniol, Claire Pagetti, and Nathanaël Sensfelder, « Identification of multicore interference », *in*: *2019 IEEE 19th International Symposium on High Assurance Systems Engineering (HASE)*, IEEE, 2019, pp. 98–106.

[24] Noureddine Bouhali et al., « Execution time modeling for cnn inference on embedded gpus », *in*: *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, 2021, pp. 59–65.

[25] Halima Bouzidi et al., « Performance prediction for convolutional neural networks on edge gpus », *in*: *Proceedings of the 18th ACM International Conference on Computing Frontiers*, 2021, pp. 54–62.

[26] Axel Brando et al., « Using Quantile Regression in Neural Networks for Contention Prediction in Multicore Processors », *in*: *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, ed. by Martina Maggio, vol. 231, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 4:1–4:25, ISBN: 978-3-95977-239-6, DOI: 10.4230/LIPIcs.ECRTS.2022.4, URL: https://drops.dagstuhl.de/opus/volltexte/2022/16321.

[27] Derek Bruening and Timothy Garnett, « Building dynamic instrumentation tools with DynamoRIO », *in*: *Proc. Int. Conf. IEEE/ACM Code Generation and Optimization*, 2013.

[28] Francisco J. Cazorla et al., « PROARTIS: Probabilistically Analyzable Real-Time Systems », *in*: *ACM Trans. Embedded Comput. Syst.* 12.*2s* (2013), 94:1–94:26.

[29] Francisco J. Cazorla et al., « PROARTIS: Probabilistically Analyzable Real-Time Systems », *in*: *ACM Trans. Embedded Comput. Syst.* 12.*2s* (2013), 94:1–94:26.

[30] Francisco J. Cazorla et al., « Probabilistic Worst-Case Timing Analysis: Taxonomy and Comprehensive Survey », *in*: *ACM Comput. Surv.* 52.*1* (2019), 14:1–14:35.

[31] Francisco J. Cazorla et al., « Probabilistic Worst-Case Timing Analysis: Taxonomy and Comprehensive Survey », *in*: *ACM Comput. Surv.* 52.*1* (2019), 14:1–14:35.

[32] Lizhong Chen, Drew Penney, and Daniel Jiménez, *AI for computer architecture: principles, practice, and prospects*, Springer, 2021.

[33] Stanley F Chen and Joshua Goodman, « An empirical study of smoothing techniques for language modeling », *in*: *Computer Speech & Language* 13.*4* (1999), pp. 359–394.

[34] Tianqi Chen and Carlos Guestrin, « Xgboost: A scalable tree boosting system », *in*: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[35] Weifan Chen et al., « Low-Overhead Online Assessment of Timely Progress as a System Commodity », *in*: *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, ed. by Alessandro V. Papadopoulos, vol. 262, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 13:1–13:26, ISBN: 978-3-95977-280-8, DOI: 10.4230/LIPIcs.ECRTS.2023.13, URL: https://drops.dagstuhl.de/opus/volltexte/2023/18042.

[36] Yiran Chen et al., « A survey of accelerator architectures for deep neural networks », *in*: *Engineering* 6.*3* (2020), pp. 264–274.

[37] Yishen Chen et al., « BHive: A benchmark suite and measurement framework for validating x86-64 basic block performance models », *in*: *2019 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2019, pp. 167–177.

[38] Zhuo Chen and Yuhong Yang, « Assessing forecast accuracy measures », *in*: *Preprint Series* (2004).

[39] Vladimir Cherkassky and Yunqian Ma, « Practical selection of SVM parameters and noise estimation for SVM regression », *in*: *Neural networks* 17.*1* (2004), pp. 113–126.

[40] Antoine Colin and Isabelle Puaut, « Worst Case Execution Time Analysis for a Processor with Branch Prediction », *in*: *Real-Time Systems* 18.*2/3* (2000), pp. 249–274.

[41] Cédric Courtaud et al., « Improving prediction accuracy of memory interferences for multicore platforms », *in*: *2019 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2019, pp. 246–259.

[42] Adele Cutler, D Richard Cutler, and John R Stevens, « Random forests », *in*: *Ensemble machine learning: Methods and applications* (2012), pp. 157–175.

[43] Zihang Dai et al., « Transformer-xl: Attentive language models beyond a fixed-length context », *in*: *arXiv preprint arXiv:1901.02860* (2019).

[44] Zihang Dai et al., « Transformer-xl: Attentive language models beyond a fixed-length context », *in*: *arXiv preprint arXiv:1901.02860* (2019).

[45] Mickaël Dardaillon et al., « Reconciling Compiler Optimizations and WCET Estimation Using Iterative Compilation », *in*: *IEEE Real-Time Systems Symposium (RTSS)*, 2019.

[46] Dibyendu Das and Sandya Mannarswamy, « ML-driven Hardware Cost Model for MLIR », *in*: *arXiv preprint arXiv:2302.11405* (2023).

[47] Dakshina Dasari et al., « Identifying the sources of unpredictability in COTS-based multicore systems », *in*: *2013 8th IEEE international symposium on industrial embedded systems (SIES)*, IEEE, 2013, pp. 39–48.

[48] Robert I Davis et al., *Impact Case Study: How long does your real-time software take to run?*, tech. rep., Technical report, University of York, 2015. URL: https://www-users. cs. york ...

[49] Arnaldo Carvalho De Melo, « The new linux'perf'tools », *in*: *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[50] Théo Degioanni and Isabelle Puaut, « StAMP: Static Analysis of Memory access Profiles for real-time tasks », *in*: *20th International Workshop on Worst-Case Execution Time Analysis (WCET 2022)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[51] Jean-François Deverge and Isabelle Puaut, « Safe measurement-based WCET estimation », *in*: *International Workshop on WCET Analysis*, 2005.

[52] 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.

[53] 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.

[54] Jacob Devlin et al., « 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 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, ed. by Jill Burstein, Christy Doran, and Thamar Solorio, Association for Computational Linguistics, 2019, pp. 4171–4186, DOI: 10.18653/v1/n19-1423, URL: https://doi.org/10.18653/v1/n19-1423.

[55] Jacob Devlin et al., « 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*, Association for Computational Linguistics, 2019, pp. 4171–4186.

[56] Jacob Devlin et al., « Bert: Pre-training of deep bidirectional transformers for language understanding », *in*: *arXiv:1810.04805* (2018).

[57] Jacob Devlin et al., « Bert: Pre-training of deep bidirectional transformers for language understanding », *in*: *arXiv preprint arXiv:1810.04805* (2018).

[58] Steven HH Ding, Benjamin CM Fung, and Philippe Charland, « Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization », *in*: *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 472–489.

[59] Boris Dreyer et al., « Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs », *in*: *International Workshop on WCET Analysis*, 2016, 4:1–4:11.

[60] Christophe Dubach et al., « Fast compiler optimisation evaluation using code-feature based performance prediction », *in*: *Proceedings of the 4th international conference on Computing frontiers*, 2007, pp. 131–142.

[61] Christof Ebert et al., « Cyclomatic complexity », *in*: *IEEE software* 33.*6* (2016), pp. 27–29.

[62] Dumitru Erhan et al., « Why does unsupervised pre-training help deep learning? », *in*: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2010, pp. 201–208.

[63] Heiko Falk and Paul Lokuciejewski, « A compiler framework for the reduction of worst-case execution times », *in*: *Real-Time Systems* 46.*2* (2010), pp. 251–300.

[64] Heiko Falk et al., « TACLeBench: A benchmark collection to support worst-case execution time research », *in*: *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[65] Zhangyin Feng et al., « Codebert: A pre-trained model for programming and natural languages », *in*: *arXiv preprint arXiv:2002.08155* (2020).

[66] Christian Ferdinand and Reinhold Heckmann, « ait: Worst-case execution time prediction by static program analysis », *in*: *Building the Information Society: IFIP 18th World Computer Congress Topical Sessions 22–27 August 2004 Toulouse, France*, Springer, 2004, pp. 377–383.

[67] Christian Ferdinand and Reinhard Wilhelm, « Efficient and Precise Cache Behavior Prediction for Real-Time Systems », *in*: *Real-Time Systems* 17.*2-3* (1999), pp. 131–181.

[68] Christian Ferdinand and Reinhard Wilhelm, « Efficient and Precise Cache Behavior Prediction for Real-Time Systems », *in*: *Real-Time Systems* 17.*2-3* (1999), pp. 131–181.

[69] Francis Galton, « Regression towards mediocrity in hereditary stature. », *in*: *The Journal of the Anthropological Institute of Great Britain and Ireland* 15 (1886), pp. 246–263.

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

[71] SEGGER Microcontroller GmbH, *Ozone User Guide & Reference Manual*, en, URL: `https://www.segger.com/`.

[72] SEGGER Microcontroller GmbH, *Ozone User Guide & Reference Manual*, en, URL: `https://www.segger.com/`.

[73] Jan Gustafsson et al., « Approximate worst-case execution time analysis for early stage embedded systems development », *in*: *Software Technologies for Embedded and Ubiquitous Systems: 7th IFIP WG 10.2 International Workshop, SEUS 2009 Newport Beach, CA, USA, November 16-18, 2009 Proceedings 7*, Springer, 2009, pp. 308–319.

[74] Jan Gustafsson et al., « 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.

[75] Matthew R Guthaus et al., « MiBench: A free, commercially representative embedded benchmark suite », *in*: *4th IEEE international workshop on workload characterization*, 2001.

[76] Laurens Haan and Ana Ferreira, *Extreme value theory: an introduction*, vol. 3, Springer, 2006.

[77] Damien Hardy and Isabelle Puaut, « WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches », *in*: *IEEE Real-Time Systems Symposium (RTSS)*, 2008, pp. 456–466.

[78] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut, « The Heptane Static Worst-Case Execution Time Estimation Tool », *in*: *International Workshop on WCET Analysis*, 2017, 8:1–8:12.

[79] 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.

[80] Simon Haykin, *Neural networks and learning machines, 3/E*, Pearson Education India, 2009.

[81] John L Hennessy and David A Patterson, *Computer architecture: a quantitative approach*, Elsevier, 2011.

[82] Sepp Hochreiter and Jürgen Schmidhuber, « Long short-term memory », *in*: *Neural computation* 9.*8* (1997), pp. 1735–1780.

[83] Sepp Hochreiter and Jürgen Schmidhuber, « Long short-term memory », *in*: *Neural computation* 9.*8* (1997), pp. 1735–1780.

[84] Arthur E Hoerl and Robert W Kennard, « Ridge regression: Biased estimation for nonorthogonal problems », *in*: *Technometrics* 12.*1* (1970), pp. 55–67.

[85] Ling Huang et al., « Predicting execution time of computer programs using sparse polynomial regression », *in*: *Advances in neural information processing systems* 23 (2010).

[86] Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx, « A New Hybrid Approach on WCET Analysis for Real-Time Systems Using Machine Learning », *in*: *International Workshop on WCET Analysis*, 2018, 5:1–5:12.

[87] 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.

[88] 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.

[89] Thomas Huybrechts et al., « COBRA-HPA: a block generating tool to perform hybrid program analysis », *in*: *International journal of grid and utility computing* 10.*2* (2019), pp. 105–118.

[90] Thomas Huybrechts et al., « 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)*, Springer, 2019, pp. 415–425.

[91] Thomas Huybrechts et al., « 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)*, Springer, 2019, pp. 415–425.

[92] Intel, *Intel® Architecture Code Analyzer*, en, URL: `https://www.intel.com/content/www/us/en/developer/articles/tool/architecture-code-analyzer.html` (visited on 12/07/2021).

[93] Natasha Jaques et al., « Tuning recurrent neural networks with reinforcement learning », *in*: *Int. conference on learning representations*, 2017.

[94] Richard Johnson, David Pearson, and Keshav Pingali, « The Program Structure Tree: Computing Control Regions in Linear Time », *in*: *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, ed. by Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa, ACM, 1994, pp. 171–185, DOI: `10.1145/178243.178258`, URL: `https://doi.org/10.1145/178243.178258`.

[95] Daniel Kästner et al., « TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis », *in*: *International Workshop on WCET Analysis*, vol. 72, OpenAccess Series in Informatics (OASIcs), 2019, 1:1–1:11, ISBN: 978-3-95977-118-4.

[96] Daniel Kästner et al., « 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.

[97] Urvashi Khandelwal et al., « Sharp nearby, fuzzy far away: How neural language models use context », *in*: *arXiv preprint arXiv:1805.04623* (2018).

[98] Diederik P. Kingma and Jimmy Ba, « Adam: A Method for Stochastic Optimization », *in*: *3rd Int. Conference on Learning Representations (ICLR), San Diego, CA, USA*, 2015.

[99] Raimund Kirner and Peter Puschner, « Discussion of misconceptions about WCET analysis », *in*: *WCET*, 2003, pp. 61–64.

[100] Raimund Kirner et al., « Using Measurements as a Complement to Static Worst-Case Execution Time Analysis », *in*: *Intelligent Systems at the Service of Mankind* 2 (Jan. 2006).

[101] Raimund Kirner et al., « Using Measurements as a Complement to Static Worst-Case Execution Time Analysis », *in*: *Intelligent Systems at the Service of Mankind* 2 (Jan. 2006).

[102] Oliver Kramer, « Unsupervised K-nearest neighbor regression », *in*: *arXiv preprint arXiv:1107.3600* (2011).

[103] Taku Kudo and John Richardson, « Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing », *in*: *arXiv preprint arXiv:1808.06226* (2018).

[104] Vikash Kumar, « An integrated approach of Genetic Algorithm and Machine Learning for generation of Worst-Case Data for Real-Time Systems », *in*: *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, IEEE, 2022, pp. 87–95.

[105] Vikash Kumar, « Estimation of an Early WCET Using Different Machine Learning Approaches », *in*: *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, Springer, 2022, pp. 297–307.

[106] Sampo Kuutti et al., « A survey of deep learning applications to autonomous vehicle control », *in*: *IEEE Transactions on Intelligent Transportation Systems* 22.*2* (2020), pp. 712–733.

[107] Sang Gyu Kwak and Jong Hae Kim, « Central limit theorem: the cornerstone of modern statistics », *in*: *Korean journal of anesthesiology* 70.*2* (2017), pp. 144–156.

[108] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann, « Pipeline Modeling for Timing Analysis », *in*: *Static Analysis, 9th International Symposium, SAS*, 2002, pp. 294–309.

[109] Jan Laukemann et al., « Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures », *in*: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018.

[110] Yongjun Lee et al., « Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with CNN », *in*: *Applied Sciences* 9.*19* (2019), p. 4086.

[111] Benjamin Lesage, Stephen Law, and Iain 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.

[112] Hanbing Li, Isabelle Puaut, and Erven 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.

[113] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra, « Modeling out-of-order processors for WCET analysis », *in*: *Real-Time Systems* 34.*3* (2006), pp. 195–227.

[114] Xianfeng Li et al., « Chronos: A timing analyzer for embedded software », *in*: *Science of Computer Programming* 69.*1-3* (2007), pp. 56–67.

[115] Xuezixiang Li, Yu Qu, and Heng Yin, « Palmtree: Learning an assembly language model for instruction embedding », *in*: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[116] Xuezixiang Li, Yu Qu, and Heng Yin, « Palmtree: Learning an assembly language model for instruction embedding », *in*: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[117] Yau-Tsun Steven Li and Sharad Malik, « Performance analysis of embedded software using implicit path enumeration », *in*: *DAC: 32nd ACM/IEEE conference on Design automation*, 1995, pp. 456–461.

[118] Yau-Tsun Steven Li and Sharad Malik, « Performance analysis of embedded software using implicit path enumeration », *in*: *DAC: 32nd ACM/IEEE conference on Design automation*, 1995, pp. 456–461.

[119] Björn Lisper, « SWEET–a tool for WCET flow analysis », *in*: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Springer, 2014, pp. 482–485.

[120] Arm Ltd, *Cycle Models*, en, URL: `https://www.arm.com/products/development-tools/simulation/cycle-models` (visited on 12/07/2021).

[121] Scott M Lundberg and Su-In Lee, « A unified approach to interpreting model predictions », *in*: *Advances in neural information processing systems* 30 (2017).

[122] Thang Luong, Richard Socher, and Christopher Manning, « Better Word Representations with Recursive Neural Networks for Morphology », *in*: *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, Sofia, Bulgaria: Association for Computational Linguistics, Aug. 2013, pp. 104–113, URL: `https://aclanthology.org/W13-3512`.

[123] Mingsong Lv et al., « A Survey on Static Cache Analysis for Real-Time Systems », *in*: *LITES* 3.*1* (2016), 05:1–05:48.

[124] Martin Maas, « A taxonomy of ML for systems problems », *in*: *IEEE Micro* 40.*05* (2020), pp. 8–16.

[125] David MacNeil and Chris Eliasmith, « Fine-tuning and the stability of recurrent neural networks », *in*: *PloS one* 6.*9* (2011).

[126] Batta Mahesh, « Machine learning algorithms-a review », *in*: *International Journal of Science and Research (IJSR).[Internet]* 9.*1* (2020), pp. 381–386.

[127] Larry R Medsker and LC Jain, « Recurrent neural networks », *in*: *Design and Applications* 5.*64-67* (2001), p. 2.

[128] Sören Meinken, *Measurement-based WCET estimation in multicore real-time systems*, 2022.

[129] Charith Mendis et al., « Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks », *in*: *Int. Conference on machine learning*, PMLR, 2019.

[130] Charith Mendis et al., « Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks », *in*: *International Conference on machine learning*, PMLR, 2019, pp. 4505–4515.

[131] Fanqi Meng, Xiaohong Su, and Zhaoyang Qu, « Nonlinear approach for estimating WCET during programming phase », *in*: *Cluster Computing* 19.*3* (2016), pp. 1449–1459.

[132] Fanqi Meng, Haochen Sun, and Jingdong Wang, « Establish Program WCET and Energy Consumption Prediction Model Based on LM Algorithm », *in*: *2021 13th International Conference on Computational Intelligence and Communication Networks (CICN)*, IEEE, 2021, pp. 86–90.

[133] Tomas Mikolov et al., « Distributed representations of words and phrases and their compositionality », *in*: *Advances in neural information processing systems* 26 (2013).

[134] Christoph Molnar, Giuseppe Casalicchio, and Bernd Bischl, « Interpretable machine learning–a brief history, state-of-the-art and challenges », *in*: *ECML PKDD 2020 Workshops: Workshops of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2020): SoGood 2020, PDFL 2020, MLCS 2020, NFMCP 2020, DINA 2020, EDML 2020, XKDD 2020 and INRA 2020, Ghent, Belgium, September 14–18, 2020, Proceedings*, Springer, 2021, pp. 417–431.

[135] Philip J Mucci et al., « PAPI: A portable interface to hardware performance counters », *in*: *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[136] Kris Nikov et al., « Robust and accurate fine-grain power models for embedded systems with no on-chip pmu », *in*: *IEEE Embedded Systems Letters* 14.*3* (2022), pp. 147–150.

[137] Arjun Panesar, *Machine learning and AI for healthcare*, Springer, 2019.

[138] Eun Jung Park, *Automatic selection of compiler optimizations using program characterization and machine learning*, University of Delaware, 2015.

[139] Fabian Pedregosa et al., « Scikit-learn: Machine Learning in Python », *in*: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[140] Ruchir Puri et al., « CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks », *in*: *arXiv preprint arXiv:2105.12655* (2021).

[141] J. Ross Quinlan, « Induction of decision trees », *in*: *Machine learning* 1 (1986), pp. 81–106.

[142] *Raspberry Pi platforms*, 2020, URL: `https://www.raspberrypi.org/`.

[143] Federico Reghenzani, Luca Santinelli, and William Fornaciari, « Dealing with uncertainty in pWCET estimations », *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 19.5 (2020), pp. 1–23.

[144] Federico Reghenzani et al., « Probabilistic-WCET reliability: On the experimental validation of EVT hypotheses », *in*: *International Conference on Omni-Layer Intelligent Systems*, 2019, pp. 229–234.

[145] Jan Reineke et al., « A definition and classification of timing anomalies », *in*: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[146] Fabian Ritter and Sebastian Hack, « Pmevo: portable inference of port mappings for out-of-order processors by evolutionary optimization », *in*: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 608–622.

[147] Younes Samih et al., « Multilingual code-switching identification via lstm recurrent neural networks », *in*: *Proceedings of the Second Workshop on Computational Approaches to Code Switching*, 2016.

[148] Luca Santinelli et al., « On the sustainability of the extreme value theory for WCET estimation », *in*: *International Workshop on WCET Analysis*, 2014.

[149] Vivek Sarkar, « Determining average program execution times and their variance », *in*: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, 1989, pp. 298–312.

[150] Arun Sathanur et al., « QuaL 2 M: Learning Quantitative Performance of Latency-Sensitive Code », *in*: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2022, pp. 913–923.

[151] Segger, *J-Trace PRO – The Leading Trace Solution*, URL: https://www.segger.com/products/debug-probes/j-trace/ (visited on 11/19/2021).

[152] Segger, *J-Trace PRO – The Leading Trace Solution*, URL: https://www.segger.com/products/debug-probes/j-trace/ (visited on 11/19/2021).

[153] Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti, « Modeling Cache Coherence to Expose », *in*: *ECRTS 2019*, 2019.

[154] Syed Abdul Baqi Shah, Muhammad Rashid, and Muhammad Arif, « Estimating WCET using prediction models to compute fitness function of a genetic algorithm », *in*: *Real-Time Systems* 56 (2020), pp. 28–63.

[155] Jun S Shim et al., « DeepPM: transformer-based power and performance prediction for energy-aware software », *in*: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022, pp. 1491–1496.

[156] Jun S Shim et al., « DeepPM: transformer-based power and performance prediction for energy-aware software », *in*: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2022, pp. 1491–1496.

[157] Stefan Stattelmann and Florian Martin, « On the Use of Context Information for Precise Measurement-Based Execution Time Estimation », *in*: *International Workshop on WCET Analysis*, 2010, pp. 64–76.

[158] D Suleiman, M Ibrahim, and I Hamarash, « Dynamic voltage frequency scaling (DVFS) for microprocessors power and energy reduction », *in*: *4th International Conference on Electrical and Electronics Engineering*, vol. 12, 2005.

[159] Ondrej Sykora et al., « GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation », *in*: *2022 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2022, pp. 14–26.

[160] Robert Tarjan, « Depth-first search and linear graph algorithms », *in*: *SIAM journal on computing* 1.*2* (1972), pp. 146–160.

[161] Stephan Thesing et al., « An abstract interpretation-based timing validation of hard real-time avionics software », *in*: *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* IEEE Computer Society, 2003, pp. 625–625.

[162] Robert Tibshirani, « Regression shrinkage and selection via the lasso », *in*: *Journal of the Royal Statistical Society Series B: Statistical Methodology* 58.*1* (1996), pp. 267–288.

[163] Philip C Treleaven et al., « Computer architectures for artificial intelligence », *in*: *Future Parallel Computers: An Advanced Course Pisa, Italy, June 9–20, 1986 Proceedings*, Springer, 2005, pp. 416–492.

[164] Ashish Vaswani et al., « Attention is all you need », *in*: *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[165] Ashish Vaswani et al., « Attention is all you need », *in*: *Advances in neural information processing systems* 30 (2017).

[166] Kapil Vaswani et al., « Microarchitecture sensitive empirical models for compiler optimizations », *in*: *Int. Symposium on Code Generation and Optimization*, IEEE, 2007.

[167] Peter Wägemann et al., « GenE: A Benchmark Generator for WCET Analysis », *in*: *International Workshop on WCET Analysis*, 2015, pp. 33–43.

[168] Jin Wang et al., « Investigating Dynamic Routing in Tree-Structured LSTM for Sentiment Analysis », *in*: *Int. Conference on Empirical Methods in Natural Language Processing and Int. Joint Conference on NLP*, 2019.

[169] Zheng Wang and Michael O'Boyle, « Machine learning in compiler optimization », *in*: *Proceedings of the IEEE* 106.*11* (2018), pp. 1879–1901.

[170] Reinhard Wilhelm et al., « The worst-case execution-time problem - overview of methods and survey of tools », *in*: *ACM Trans. Embedded Comput. Syst.* 7.*3* (2008), 36:1–36:53.

[171] Reinhard Wilhelm et al., « The worst-case execution-time problem - overview of methods and survey of tools », *in*: *ACM Trans. Embedded Comput. Syst.* 7.*3* (2008), 36:1–36:53.

[172] Reinhard Wilhelm et al., « The worst-case execution-time problem—overview of methods and survey of tools », *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 7.*3* (2008), pp. 1–53.

[173] Zonghan Wu et al., « A comprehensive survey on graph neural networks », *in*: *IEEE transactions on neural networks and learning systems* 32.*1* (2020), pp. 4–24.

[174] Tomofumi Yuki, « Understanding polybench/c 3.2 kernels », *in*: *International workshop on polyhedral compilation techniques (IMPACT)*, 2014, pp. 1–5.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : l'apprentissage machine pour l'estimation du temps

Mot clés : Processeur complexe, Temps d'exécution pire/moyen cas, Apprentissage automatique

Résumé : L'estimation du temps d'exécution des programmes est une tâche clé mais difficile, rendue encore plus complexe par la croissance de la complexité et l'insuffisance de la documentation des architectures de processeurs modernes. Bien que les méthodes traditionnelles comme les simulateurs précis au cycle soient exactes, elles sont également longues et nécessitent une compréhension approfondie de l'architecture du processeur. Pour aborder ces limitations, une nouvelle approche basée sur les données et utilisant des techniques d'apprentissage automatique a été développée. Cependant, bien que les modèles d'apprentissage automatique existants offrent des estimations rapides, ils sont principalement adaptés à des architectures simples avec des temps d'instruction constants. Ce document vise à développer de nouvelles méthodes d'apprentissage automatique pour des processeurs complexes et non documentés en introduisant la prise en compte du contexte dans les modèles de timing basés sur l'apprentissage automatique. Une approche novatrice traitant les séquences d'instructions comme un langage naturel et emploie des algorithmes d'apprentissage automatique avancés tels que les réseaux Long Short-Term Memory et les Transformers. Ceci permet au modèle de prendre en compte des caractéristiques complexes telles que les effets de cache et de pipeline, améliorant la précision pour les temps d'exécution moyens et pires cas.

Title: Machine Learning for timing estimation

Keywords: Complex processor, Worst/Average-case-execution time, Machine learning

Abstract: Estimating program execution time is a key but challenging task, further complicated by the growing complexity and insufficient documentation of modern processor architectures. While traditional methods like cycle-accurate simulators are precise, they are time-consuming and demand an in-depth understanding of the processor's architecture. A new data-driven approach utilizing machine learning techniques has been developed to address these limitations. However, while existing machine learning models offer rapid estimations, they are primarily tailored for simpler architectures with constant instruction timings. This document aims to develop new machine-learning methods for complex, undocumented processors by introducing context awareness into timing models based on machine learning. A novel approach treats instruction sequences like natural language and employs advanced machine learning algorithms such as Long Short-Term Memory networks and Transformers. This allows the model to consider complex features such as cache and pipeline effects, improving the accuracy for both worst-case and average-case execution times.