Work in Progress: Early Timing Prediction of Real-time Tasks in Continuous Integration Environments

Pengcheng Huang, Balz Maag, Thanikesavan Sivanthi and Chunwei Xing
ABB Corporate Research, Switzerland
Email: firstname.lastname@ch.abb.com

***Abstract*—Timing prediction for real-time systems, especially those based on multi-core processor technology, presents enormous challenges in the design of modern industrial control systems. Classical timing analysis techniques that have been effective in the past cannot yet keep up with the increased complexity of industrial control systems. Therefore, using hardware-in-the-loop testing to understand the timing behavior of real-time tasks is a prevalent practice across many industries. However, applying this state-of-the-practice to Continuous Integration (CI) software development workflows is expensive, and frequently leads to delayed developer feedback on task timing for code commits.**

To address this challenge, we propose the *Chronos* framework, which focuses on improving development efficiency of industrial control system software. *Chronos* utilizes CI data from both simulation and hardware-in-the-loop testing to build machine learning based, cross-platform timing prediction models, which correlate the simulated performance of the tasks with their actual timing observed on the target embedded hardware. For any new code that is committed, the timing prediction is triggered by the CI server with the trained machine learning models, enabling fast feedback on timing behavior of the committed code. We demonstrate the effectiveness of *Chronos* with preliminary results on real industrial control system setups.

I. Introduction

Execution time estimation is crucial in the development of real-time software; existing methods can be divided into three main categories: static timing analysis, dynamic timing analysis (measurement-based), and hybrid timing analysis [1]. Static timing analysis employs software and abstract processor models to determine task timing ranges, while dynamic timing analysis uses real-time system testing or active benchmarking to empirically estimate task timing. Hybrid timing analysis combines benchmarked timing on fundamental block level and program path analysis to determine task execution times.

As industrial control systems shift towards multi-core architectures and embrace Continuous Integration (CI) in real-time software development, existing timing estimation methods face challenges. Common interference channels in a multi-core environment, such as shared last level cache, bus, and memory, significantly impact task timing due to co-running tasks on the same and the other cores. This complexity makes static timing analysis considerably more complex; consequently, it becomes a common practice to adopt dynamic timing analysis in many industries. However, dynamic timing analysis through hardware-in-the-loop (HIL) testing is time-consuming, the developers may have to wait days or weeks before learning about the timing behavior of the developed code, thus, slowing down the development of real-time control software.

Commercial products, such as [2] and [3], offer solutions for obtaining feedback on task timing during continuous software development. These solutions focus on CI-triggered worst-case execution time analysis using conventional techniques. Some prior works, like [4], [5], [6], [7], [8], employ a data-driven approach for timing prediction without specifically addressing CI environments. These approaches benchmark a real-time task in both a host environment (e.g., simulation computer) and on the target embedded system; they further train a machine learning model to predict real-time software performance on the target based on host performance data. Other machine learning based approaches [9], [10] correlate analyzed program features with actual task timing estimates; this, however, requires separate tools for static code analysis if applied in a CI setup.

We propose integrating machine learning-based cross-platform timing prediction methods into the CI environment for real-time software development. To this end, we leverage the data collected from simulation-based integration tests that are run in the CI's host environment as well as the data collected from target testing using HIL. The data is continuously collected to train machine learning models correlating simulated performance of the tasks with their actual timing on the target. Subsequently, for any newly committed code, the trained models can provide an early timing prediction on the target task timing. This approach can offer faster timing feedback and would be more scalable compared to traditional methods, requiring no expert tools or knowledge about the underlying hardware. This, however, is not to replace extensive testing on a real embedded platform, which remains essential for validating timing behaviors.

To demonstrate data-driven early feedback on task timing in an industrial software development process, we introduce *Chronos* - a developers' real-time framework for early timing prediction. *Chronos* facilitates automatic testing of real-time tasks on both simulation and real target hardware platforms, continuous performance data collection, and machine learning model training to predict task execution times based on observed performance during simulations. The initial findings using *Chronos* in an industrial setting clearly demonstrate its effectiveness in providing early feedback on task timing.

2642-7346/24/$31.00 ©2024 IEEE
DOI 10.1109/RTAS61025.2024.00039

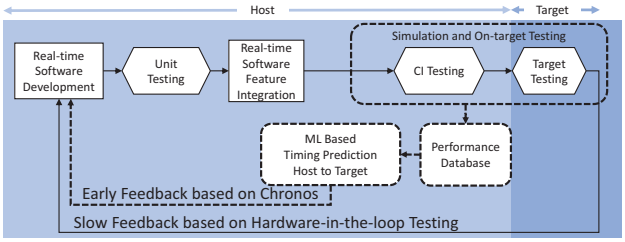


Fig. 1: *Chronos* framework integrated in a typical real-time software development workflow

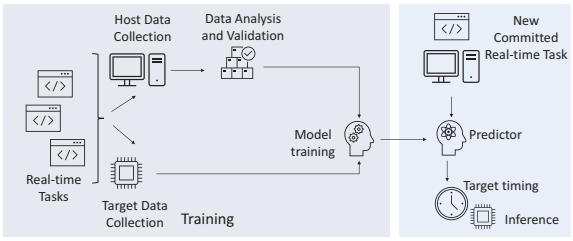


Fig. 2: Training and inference of cross-platform timing prediction models

The rest of the paper is organized as follows. Section II gives an overview of the *Chronos* framework with its main components. Section III explains machine learning based timing prediction in *Chronos*. Section IV presents the experiment results using benchmark programs. Section V summarizes this paper and outlines directions for future work.

II. *Chronos* Overview

We differentiate the developing and testing environment into host and target: A host is a developer machine or continuous integration (CI) server running simulations (CI tests) of a real-time control software. A target is the real-time embedded system which executes the real-time tasks. We note that this represents a typical setup in a test-driven industrial software development environment, where on-target and simulation-based integration tests are usually well-defined. Based on our experience, HIL tests can take days or even weeks to complete, whereas simulations can be completed in a matter of minutes or hours. For this reason, the target tests are conducted less frequent than the simulation-based integration tests.

In such a development environment, to facilitate early timing prediction, we propose *Chronos* (dotted components in Figure 1). During simulation-based integration tests, *Chronos* monitors the execution time of the real-time task being tested while logging simultaneously performance events, such as instructions completed, cache hits, and misses, that can help to characterize the timing of the task. Those performance events are used to build a machine learning based prediction model for estimating task execution time on the end target. To realize such monitoring, we adopted PAPI [11] for performance related software instrumentation. Formally, for each task τ_i, we collect the performance data sets D_i^{host} and D_i^{target} on both host and target during the simulation and HIL tests. Each data entry in the collected data set D_i^{host} corresponds to K performance related events (data features) $d = \{\text{perf}_1, \text{perf}_2, ..., \text{perf}_K\}$, $d \in D_i^{\text{host}}$. The events contain timing related hardware events as well as the task execution time. The events are selected based on our analysis in Section III. We assume perf_K is the monitored task execution time and alternatively denote it as perf_{et}. Note that, only execution time on the target is essential for correlating performance on the host to that on the target. Therefore, monitoring other hardware performance events on the target is not necessary. To summarize, we collect a host performance data set $\mathbf{D}^{\text{host}} = \{D_i^{\text{host}} \mid 1 \leq i \leq N\}$ and a target performance data set $\mathbf{D}^{\text{target}} = \{D_i^{\text{target}} \mid 1 \leq i \leq N\}$, where N is the number of real-time tasks. $\mathbf{D}^{\text{host}}$ is updated whenever host simulation is performed while $\mathbf{D}^{\text{target}}$ is updated each time target testing is done.

The machine learning model of *Chronos* can be updated, i.e. (re-)trained, as soon as the target testing is finished such that the model also takes into account the accuracy of the predicted versus actual timing (Figure 2). Note that, one can just rely on existing simulation and on-target tests to generate performance data for training the machine learning prediction model. As the CI process iterates, the training dataset will be naturally expanded with performance data for recent code commits, which could help to train better prediction models. Nevertheless, if needed, dedicated tests can be also added to construct a high quality training dataset.

Chronos can be easily integrated into the CI server (e.g. GitLab or Jenkins) and can be triggered automatically upon code commits. This would help developers to take corrective action early in the development cycle by providing quick feedback on timing prediction. Additionally, the software developers do not need to directly interface with *Chronos* because real-time prediction is manifested as test results in the CI server subsequent to code commits.

III. Cross-platform Timing Prediction in *Chronos*

The following stages describe in detail how the cross-platform timing prediction model is built using $\mathbf{D}^{\text{host}}$ and $\mathbf{D}^{\text{target}}$ corresponding to real-time benchmark programs [12].

A. Performance Data Pre-processing

Similar to common machine learning pipelines, data cleaning, analysis and feature selection are performed before building the actual machine learning model. Specifically, we clean the data to remove any entries with corrupted data (zero monitored timing or NA values) due to failed test run. To visually inspect the data, we plot the cleaned data and examine the correlation between collected features to gain understanding about the data. To further inspect data from different real-time tasks, we utilize Principal Component Analysis (PCA) [13] to reduce data dimensionalities and visually understand similarities/differences among performance data of different real-time tasks. Figure 3 shows the PCA results of the host data. We can observe that most of the tasks form a cluster and, hence, behave similarly in terms of their performance events and execution time. Some tasks however are clear outliers, which indicates a significantly different behavior compared to

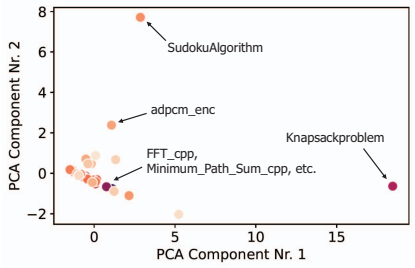
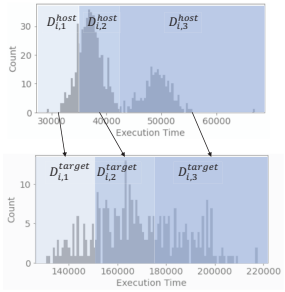


Fig. 3: PCA of host data, where each point represents a benchmark task; some tasks show considerably different behavior compared to the majority of tasks



$(\pi_{host}(D_{i,1}^{host}) \rightarrow \pi_{target}(D_{i,1}^{target}))$

$(\pi_{host}(D_{i,2}^{host}) \rightarrow \pi_{target}(D_{i,2}^{target}))$

$(\pi_{host}(D_{i,3}^{host}) \rightarrow \pi_{target}(D_{i,3}^{target}))$

Fig. 4: Percentile window based cross-platform timing prediction by correlating statistics on host to that on target

the majority of the benchmarked tasks. This result also has an impact on the overall prediction accuracy and served as inspiration for our tailored model presented in Section III-C.

One critical problem at this stage is to find good features (performance events) on the host, which can later help to predict the actual timing on the target. To solve this, we applied the *Searching for Uncorrelated List of Variables (SULOV)* method [14] to select features that are uncorrelated with each other and correlated with the target execution time.

B. Percentile Window Based Host to Target Timing Prediction

It should be noted that there is no one-to-one relationship between the traces on the host and the target. Some prior research works try to fix same task input on both host and target and assume traces on host and target are correlated this way [5]. Others propose relatively complex statistical methods to find the alignment between host and target traces by finding an alignment that maximizes correlation [6]. However, it is still arguable whether such alignments on trace level do exist in reality. We instead propose an alternative based on percentile window alignment, see Figure 4. The idea is to align statistics in the form of percentile windows on host to target rather than individual data points (or traces). The intuition is that worst-case (best-case) timing behaviours on the host should statistically help to explain worst-case (best-case) timing behaviours on the target and we generalize this with our percentile window based problem formulation. In detail, for data collected on both host and target for the same real-time task, we group them according to their host and target timing distributions. Assume for a real-time task τ_i we divide its host and target data each into W windows/groups. We take its host data $D_i^{\text{host}} = \{D_{i,w}^{\text{host}} \mid 1 \leq w \leq W\}$ as an example and note the w_{th} window with $D_{i,w}^{\text{host}} = \{d \mid d \in D_i^{\text{host}} \wedge \ t = \text{perf}_{\text{et}} \in d \wedge \mathbb{P}_{\eta_{i,w^-}^{\text{host}}} < t \leq \mathbb{P}_{\eta_{i,w^+}^{\text{host}}}\}$. Here, $\mathbb{P}_\eta$ is used to represent the percentile of task τ_i's execution time as observed in D_i^{host}, where the percentage is η. We have consecutive percentile windows defined in this way, i.e., $\eta_{i,w^+}^{\text{host}} = \eta_{i,w+1^-}^{\text{host}}, \forall \tau_i \in \mathbf{T} \wedge 1 \leq w \leq W$ and $\{\eta_{i,w^+} \mid 1 \leq w \leq W\}$ is an increasing sequence with $\eta_{i,W^+}^{\text{host}} = 100$ and $\eta_{i,1^-}^{\text{host}} = 0$. Similarly the target performance data of task τ_i is divided into percentile windows $\{D_{i,w}^{\text{target}} \mid 1 \leq w \leq W\}$.

Our machine learning based timing prediction from host performance events to target timing is formulated as follows:

$$\min_{\theta} \sum_{i=1}^{N_{\text{train}}} \sum_{w=1}^{W} l(f(\pi_{\text{host}}(D_{i,w}^{\text{host}}), \theta), \pi_{\text{target}}(D_{i,w}^{\text{target}})) \qquad (1)$$

Here, π represents our transformation of data points in one percentile window to a single data point. The average or maximum operation can be used for this transformation, depending on the specific machine learning problem setup. f represents a selected machine learning regression model parameterized with θ and l represents the loss function for machine learning training, which depends on the selected model and regularization methods. For regression problems, l typically contains the data loss component formulated as the squared error between timing prediction and actual target timing and regularization component such as used in ridge or lasso regressions, see details in e.g. [15], [16]. The training process minimizes the total loss and finds the best model parameters θ. We adopt and compare a list of different regression models for timing prediction, such as linear regression, lasso, elastic net, gradient boosting, decision trees and k-nearest neighbors. Note that in our machine learning formulation we are primarily interested in predicting target timing (perf_{et}). Note additionally that a standard train and test data split is performed such that N_{train} out of N real-time tasks are used for training the prediction model while the rest of the tasks are held out for testing.

C. Customization of Cross Platform Timing Prediction

The percentile window-based cross-platform prediction is designed for a universal prediction model across all programs. However, due to variations among real-time programs (refer to Figure 3), employing such a generic model might not be optimal. Training with a vastly different program could adversely affect predictions for dissimilar programs. Consequently, when predicting the timing behavior of a specific program on the target platform, it is beneficial to identify a set of training programs that closely resemble the target program, aiding in explaining its timing on the target. This approach is particularly useful in a CI environment, where we want to base our prediction for a particular commit on specifically selected data from previous commits or application versions.

Based on this insight, we developed a tailored prediction algorithm. The initial step involves sorting real-time tasks

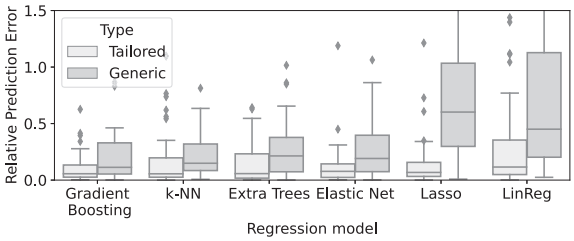


Fig. 5: Relative prediction error of different regression models

by increasing distance to the task τ, for which a timing prediction is to be made. This sorting is accomplished by measuring the distance between the centers of their host data sets, with data set center calculated as the 50th percentiles of individual features. Subsequently, a greedy forward selection is performed to include new programs in training the prediction model for τ, starting with the one closest (most similar) to τ.

IV. Experiments and Results

Setup. The following experiment and corresponding results are all based on data collected by our *Chronos* framework. We use a Linux machine as CI host, which features an octacore Intel(R) Xeon(R) Silver 4208 CPU. The target machine is a Toradex IMX 8 MPSoC based on a NXP imx8 quadmax CPU. Overall we benchmarked 48 different benchmark tasks on both the host and target machine and collected approximately 840k samples. Apart from publicly available benchmarks [12], [17], we also test one specific industrial motor control application. For each benchmark task, we divide the collected data into 10 equally-spaced percentile windows and align corresponding windows on the host and target as described in Section III-B. We then perform a leave-one-task-out cross-validation with all benchmark tasks and evaluate the performance of the different regression models in terms of how accurate we are able to predict the execution time of the task-under-test. Additionally, we compare the two different types of prediction models, i.e. a generic model trained on all available data vs. our tailored predictor trained on customized data (Sec. III-C).

Results. Figure 5 summarizes the performance over all benchmark programs. We observe a generally well performing prediction accuracy with a median error between 5 and 11% of our tailored predictor and a clear improvement over the generic approach, i.e. 2x lower error in average. The median accuracy of the specific control application was in line with the other benchmarks and between 5 and 20% error depending on the regression model. However there are also some clear outliers with errors above 100% even for the tailored predictor. These are caused by programs, which behave significantly different than the majority of other programs. This shows that it is important to collect sufficient representative data from tasks that are similar to the predicted task. As mentioned earlier, in theory this can be easily done in a CI environment using data from previous commits and, therefore, we will carefully evaluate this in future work. Additionally, we also plan to evaluate the prediction performance for situations where the task-under-test is deployed in a multi-core environment and scheduled together with other real-time tasks.

V. Conclusion and Future Work

This paper presents the *Chronos* framework, enabling early timing prediction in continuous integration (CI) for real-time software development. Leveraging host simulation and on-target tests, *Chronos* continuously collects performance data, with which machine learning models are trained to correlate host performance events during simulation with target execution times. This allows quick timing estimation immediately after host simulation without the need to wait for time-consuming tests on real target hardware. Illustrated with open benchmarks and an industrial application, promising initial results are demonstrated. Integrating *Chronos* in actual development teams' CI environments is currently work in progress for a more comprehensive evaluation of the framework.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[2] "Absint," https://www.absint.com/, accessed: 2024-01-17.

[3] "Bound-t," http://www.tidorum.fi/en/index.html, accessed: 2024-01-17.

[4] I. Baldini, S. J. Fink, and E. Altman, "Predicting gpu performance from cpu runs using machine learning," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, 2014, pp. 254–261.

[5] X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer, "Learning-based analytical cross-platform performance prediction," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2015, pp. 52–59.

[6] X. Zheng, H. Vikalo, S. Song, L. K. John, and A. Gerstlauer, "Sampling-based binary-level cross-platform performance estimation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1709–1714.

[7] Y. Kim, P. Mercati, A. More, E. Shriver, and T. Rosing, "P4: Phase-based power/performance prediction of heterogeneous systems via neural networks," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 683–690.

[8] K. O'Neal, M. Liu, H. Tang, A. Kalantar, K. DeRenard, and P. Brisk, "Hlspredict: Cross platform performance prediction for fpga high-level synthesis," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[9] A. Bonenfant, D. Claraz, M. de Michiel, and P. Sotin, "Early WCET Prediction Using Machine Learning," in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, 2017.

[10] P. Altenbernd, J. Gustafsson, B. Lisper, and F. Stappert, "Early execution time-estimation through automatically generated timing models," 2016.

[11] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[12] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research." Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.

[13] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, 1987.

[14] "featurewiz: New python library for fast feature selection," https://github.com/AutoViML/featurewiz, accessed: 2023-02-22.

[15] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[16] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[17] "Acm icpc algorithms," https://github.com/matthewsamuel95/ACM-ICPC-Algorithms, accessed: 2023-02-22.