

# Boosting Embedded System Development: A Case for Rapid Testing

1<sup>st</sup> Rainer Poisel  
Software Development  
honeytreeLabs Cooperation  
St. Poelten, Austria  
rainer@honeytreelabs.com

2<sup>nd</sup> Stefan Riegler  
Cyber Security  
honeytreeLabs Cooperation  
St. Poelten, Austria  
stefan@honeytreelabs.com

**Abstract**—In software development, developers’ time is precious, with bottlenecks often occurring during lengthy integration and system tests with hardware interactions. These prolonged test cycles frustrate developers and hamper productivity. Teams sometimes limit tests to a subset (smoke testing), risking bug oversights. We aim to expedite testing by utilizing additional devices. While seemingly straightforward, this approach poses unique challenges we will discuss in this article. Our objective is to enable developers to focus on core tasks, mitigating delays from lengthy continuous integration runs.

We present an algorithmic solution for parallelization, where scalability efforts can be reduced to adding more devices while maintaining correctness and reliability despite increased speed. We delve into test optimization strategies, in the context of Hardware-in-the-Loop (HiL) testing. To illustrate practicality, we implement these concepts within established frameworks such as pytest and labgrid. Our approach accelerates automated testing by connecting extra Devices under Test (DUT), reducing continuous integration run-times by 60% and test execution times by 85%.

**Keywords**—software testing; test execution performance; cutting stock problem; pytest, labgrid

## I. INTRODUCTION AND STATE OF THE ART

Software testing, an essential component for ensuring high-quality software, involves key aspects such as integration and system testing. While tests based on using actual parts of the system in real-world scenarios are often more time-consuming than unit tests, they are critical in assessing module interactions and overall system performance. Their thorough nature, although lengthier, is pivotal in ensuring that the software meets its intended requirements, thereby solidifying their role in software quality assurance. The reduction of test execution times in software development is of paramount importance. It enables quicker feedback for developers, which leads to faster issue resolution and ultimately enhances the quality of the software. This increase in efficiency not only boosts productivity but also enhances cost-effectiveness. This is particularly crucial in rapid development cycles where a swift time-to-market is a

key factor. Moreover, faster test execution supports better continuous integration practices, increases test coverage, facilitates test-driven development, and improves resource utilization. It also positively impacts developer morale by reducing downtime and frustration. Consequently, optimizing test performance is essential for maintaining a competitive advantage in the dynamic software industry, where the ability to deliver high-quality software quickly can be a significant differentiator.

Existing research on the execution performance of test execution in software development can be divided into several groups:

- test case selection and prioritization
- improving the execution performance of test sets
- parallel execution of tests in existing test frameworks

In this paper, we focus on improving the execution performance of a known set of tests by executing them in parallel.

### A. Test Case Selection and Prioritization

Scientific research focusing on test case selection and prioritization to improve the performance of running automated tests covers multiple strategies to achieve that goal. The paper “The Art of Testing Less without Sacrificing Quality” [15] presents THEO, a cost-based test selection strategy aimed at optimizing software testing processes. THEO dynamically decides to skip test executions when the expected cost of running a test exceeds the cost of not running it. The approach ensures that all tests execute on all code changes at least once before shipping, thereby maintaining product quality. The strategy was evaluated on three major Microsoft products, leading to a significant reduction in test executions (up to 50%) and test time (up to 47%), while still catching defects effectively. This resulted in considerable cost savings, up to \$2 million per development year per product. The paper confirms that THEO can enhance testing efficiency without compromising product quality.

In his master’s thesis [11], Cheruiyot diverges from traditional test execution methods. It introduces a machine learning-driven approach for selective regression testing, aimed at

optimizing test case selection rather than the test execution performance. By employing natural language processing and machine learning models, this research not only streamlines the process but also significantly reduces the dependence on manual expertise.

A study on improving CI strategies [16] investigates 14 variants of 10 CI-improving techniques and suggests a hybrid approach between computational-cost reduction, missed failure observation, and time-to-feedback reduction. While not mentioning strategies how to increase the performance of actually run tests, the authors propose to focus on test-prioritization in order to achieve better advancement in the observation of test failures, highlighting their effectiveness in reducing time-to-feedback. Najafi et al. [18] explain the importance of considering test execution history. The study adopts test selection and prioritization techniques based on historical test failure frequency, association, and testing costs. As a result, the authors found that proposed approaches can significantly reduce test execution time, with variations in effectiveness based on the specific approach used.

### B. Improving Executing Performance of Test Sets

Executing software tests could be considered a typical workload for parallel computing, akin to any other workload suitable for parallel computation. In this paper, we examine the nuances of scaling the workload associated with executing software tests, specifically differentiating between vertical and horizontal scaling approaches. Vertical scaling refers to augmenting the capabilities of an existing system, such as enhancing CPU or RAM capacities. However, our focus is not on this approach. Instead, this paper centers on horizontal scaling, which involves distributing the workload across multiple so-called worker nodes.

In the realm of high-performance computing, the pertinent sub-topic is known as *task parallelism* or *task-parallel programming* [12]. Various fundamental models of task parallelism are pivotal in defining the methodologies for decomposing a given workload among multiple workers [14], [17]. Within the context of executing software tests, the individual tests represent discrete tasks to be processed by the system. These tasks often exhibit significant variation in size and complexity, rendering only certain models of task parallelism suitable for this specific challenge. In our study, we specifically explored the efficacy of the *work pool model* and the *fork-join model* in addressing the unique demands of software test execution.

We identified a significant limitation of the *work pool model* in this context. The inherent issue arises when a complex or long-lasting test is allocated to a worker node towards the end of test execution processes. In such scenarios, this imbalance often leads to a situation where the majority of worker nodes remain idle, waiting for the completion of the long-lasting test by a single node. This inefficiency in resource utilization underlines a critical drawback of the work pool model for our specific use case. Consequently, our attention shifts to the *fork-join model*, which presents a more promising approach for this application. The *fork-join model* necessitates a strategic

distribution of tests, ensuring that each worker node is assigned tasks in a manner that maximizes efficiency and minimizes idle time. This requires a meticulous balancing of test loads, ensuring that they are divided in a way that aligns with the capabilities and availability of the given worker nodes. Such an approach promises to optimize the parallel execution of software tests, leveraging the strengths of the *fork-join model* to address the challenges we have identified.

### C. Parallelism in Existing Test Frameworks

The JUnit 5 [2] test framework, allows for simple approaches to parallel execution of tests. These approaches do not consider any weights or optimal distribution of the workload to involved worker nodes. While not being able to assign the execution of specific tests to given nodes, it is possible to influence the test execution order by implementing so-called *Orderer* interfaces.

To facilitate parallel test execution in pytest [4], plugins like *pytest-parallel* [5] have been introduced. However, the *pytest-parallel* plugin, as of now, is no longer actively maintained. Moreover, a significant limitation of this plugin is its approach to test distribution; it does not consider the individual execution times of tests. This oversight can lead to inefficient test distribution among executors, potentially negating the benefits of parallel execution by causing imbalances in workload and underutilization of resources. In this article, we will focus on how we can achieve an even distribution of tests to multiple workers.

The *pytest-xdist* plugin for pytest [6] improves parallel test execution by taking different test runtimes into account. Unlike the method described in our article, it dynamically re-evaluates and reassigns tests among workers to optimize execution efficiency, a feature not present in our approach. This continuous redistribution leads to better resource utilization and more efficient testing processes compared to static test distribution methods.

Utilizing the pytest [4] and labgrid [9] frameworks, we will present a series of practical examples. These examples are carefully designed to not only illustrate the methodologies and best practices within these frameworks but also to provide concrete evidence of their relevance in practical scenarios.

## II. IMPROVING TEST EXECUTION TIME

Fig. 1 illustrates the two phases of test runs. The first phase involves the collection process, during which all source units in the project are assessed for tests. Each test to be executed is represented as so called test item. The outcome of this phase is a collection of test items that are subsequently executed in the second phase known as the execution phase.



Fig. 1. Test Execution Phases

In its original implementation, the pytest framework conducts tests in sequential order. However, the pytest framework allows for hooking into different stages of the test execution process by registering callback functions. These callback functions allow to, e.g. change the test execution order, filter tests, or attach additional attributes to collected test items. In our efforts to speed up we want to build on these possibilities to allow for parallel execution of the desired tests.

### A. Distributing Tests to Multiple Workers

Our objective is the concurrent execution of test items by several test executors. To achieve this, we need to introduce an additional step between the first and the second one: distributing tests to multiple workers as shown in Fig. 2.

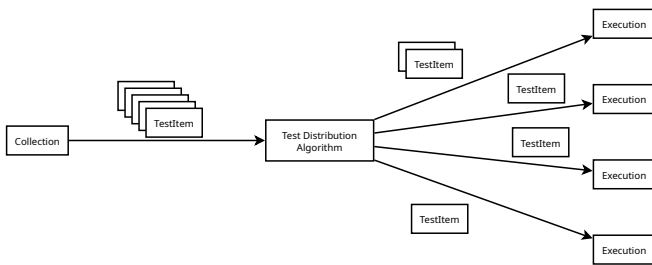


Fig. 2. pytest Phases of Parallel Execution

In addressing the optimization of test execution performance, we initially considered the worker pool model, wherein a controller dynamically assigns tasks to worker nodes based on their availability. This model promises efficiency by continually monitoring the workload of each worker and allocating new tasks as soon as a worker becomes idle. However, this method presents potential pitfalls in terms of optimal test item distribution. A critical challenge lies in the inherent limitation of this approach to accurately align with the intricacies of specific test frameworks, such as pytest, particularly during their collection-execution phases. A fundamental requirement for enhancing test execution performance in a distributed environment is the equitable distribution of tests across multiple workers. This necessitates a strategy wherein each worker is assigned tests in a manner that balances the total execution time as uniformly as possible across all nodes. As mentioned above, the simplistic nature of the worker pool model often falls short in this aspect, leading to an uneven workload distribution where some workers may be idle while others are overburdened. To address the identified limitations of the worker pool model, particularly its inadequacy in achieving a balanced distribution of test execution workload, we adopted a proactive approach in allocating tasks to worker nodes. This strategy involves pre-determining the distribution of test items to each worker node prior to initiating the actual test execution step. The intricate details of the algorithms employed, along with the rationale behind their selection and the methodologies used for their implementation, are comprehensively discussed in the next section. This section then provides a deeper insight into the algorithmic foundations of our approach and illustrates how these algorithms contribute

to the overarching goal of enhancing test execution performance in a distributed testing environment.

In the development of our test execution performance improvement methodology, a pivotal consideration was adherence to the official, public pytest API. This strategic decision was guided by the intent to maximize the future-proofing of our approach. As of February 2024, the pytest framework continues to evolve rapidly, characterized by frequent updates, with multiple commits per day, and major releases approximately every two years. This dynamic nature of the framework necessitates a solution that is both adaptable and resilient to future changes.

Conforming exclusively to the official pytest API imposed certain constraints, particularly in terms of adhering to the predefined collection and execution phases inherent to the framework. To integrate an additional test item distribution step within these constraints, we devised a methodology that involved the separate execution of the collection and execution phases which is possible using only the public pytest API. This separation was instrumental in facilitating the distribution of test items across different nodes. However, this approach introduced an additional complexity: the need to make pytest test items serializable. The rationale behind this requirement stems from the fact that the collection and execution phases are conducted in distinct contexts such as processes or individual worker nodes. Serialization of test items thus became a critical step to enable their transfer between these contexts, ensuring a seamless and efficient distribution.

### B. Achieving Even Distribution Among Multiple Workers

Our approach conceptualizes the distribution challenge as a one-dimensional cutting stock problem (1DCSP). In this paradigm, the execution time of tests is analogous to 'length' or 'weights' in 1DCSP terminology. This reframing allows us to leverage the extensive research already conducted in this area, such as the methodologies and algorithms developed and discussed in works like Bertolini et al. [10]. The 1DCSP framework offers a robust foundation for developing a more nuanced and effective distribution strategy, ensuring a more balanced and efficient allocation of test items across the worker nodes.

We have implemented a simplified variant of the cutting-stock algorithm, aiming to reduce dependencies on external libraries. Detailed in Listing 1, our algorithm adeptly handles both weighted and unweighted test items, a feature that facilitates the step-by-step optimization of existing codebases for parallel executability. Initially, tests are distributed evenly among workers by number; however, by integrating weighting annotations, we ensure a more equitable distribution of the total test execution time among available workers. This flexibility in handling both test item types is crucial for gradual optimization towards parallel execution.

The algorithm's design is tailored for distributing software test items across multiple worker nodes, thereby optimizing test execution performance in parallel computing environments. It accepts inputs including a dictionary of Python modules mapped to the test items in that Python module, a

`num_workers` parameter to determine the maximum number of worker nodes for the test run (defaulted to 1), and a `split_threshold` parameter that influences the division of module tests among workers. The initialization creates a series of buckets, each for an individual worker, into which test items are distributed based on their weighted execution times or, for unweighted items, in a round-robin fashion. This methodology not only ensures an efficient workload distribution but also maximizes the performance of each worker node, effectively addressing the challenges of parallel test execution.

In optimizing test execution performance, the flexibility in defining execution weights is a critical factor. While one could employ a limited set of predefined weights, such as numerical tiers (e.g., 1, 2, 3) or color codes (e.g., green, yellow, red), this approach inherently lacks precision. Limiting the range of execution weights to a predefined set can lead to less accurate representations of each test’s resource demands, potentially resulting in suboptimal distribution of test tasks. In contrast, by assigning estimated execution times to each test, we provide a more granular and realistic measure of resource requirements. This method not only aligns better with real-world scenarios but also enhances performance by allowing for a more accurate and efficient allocation of computational resources across testing tasks.

In our current implementation, we determine the execution time of test items empirically by manually triggering test runs and extracting the run times from execution logs. The Python snippet in Listing 2 illustrates how to assign these execution weights using the `pytest` framework: test functions are annotated with a custom decorator, `@pytest.mark.est_exec_duration`, which assigns an estimated execution duration to the test. For instance, the `test_configure_parameters` function is marked with an estimated execution duration of 17 units (which could represent any relevant time metric; we are using seconds). This annotation allows our test distribution algorithm to consider the execution weight of each test, leading to a more efficient allocation of test cases across available testing resources. By quantifying the execution time in this manner, we significantly improve the precision of workload distribution among worker nodes, thereby optimizing overall test execution efficiency.

```

1 @pytest.mark.est_exec_duration(17)
2 def test_configure_parameters(...):
3     ...

```

Listing 2. Sample weight assignment

### C. The Big Picture

In the process of conducting integration and system tests, considerable attention must also be given to the phases of preparation and cleanup, alongside the actual execution of test items. Initially, this involves allocating Devices Under Test (DUTs) from a pool of available hardware and configuring them with the necessary software, such as specific firmware versions. These preparatory steps are crucial for ensuring the tests are conducted under consistent and controlled conditions.

Following the test execution, there is a cleanup phase where the DUTs are released back into the pool, making them available for future test runs. This phase is essential for maintaining the efficiency of the testing infrastructure. The entire workflow, from device allocation to release, is detailed in Fig. 3, providing an overview of the testing process that is integral to understanding and improving test execution performance.

In our pursuit to optimize test execution performance, we have identified potential areas of enhancement within the preparation and cleanup phases. To this end, we have developed a refined activity diagram, detailed in Fig. 4, which illustrates the process outlined in Fig. 3 in more detail. This diagram is informed by our use of the `pytest` and `labgrid` frameworks and delineates the sequential and parallel activities involved in the testing process.

The initial phase of the testing process entails the sequential reservation of the required Devices Under Test (DUTs). If the reservation duration surpasses a predefined timeout, the test starts with the number of DUTs reserved up to that point; absent a set timeout, the system continues to attempt reservation of at least one DUT indefinitely. Following this, we proceed to the setup phase, where the DUTs are prepared for testing. This involves flashing the relevant firmware version onto the devices, a step that can be notably time-consuming, especially when dealing with large firmware images and the inherent limitations of flash memory write speeds. Concurrently with the first two steps, we engage in a parallel activity: the analysis of previous test runs to ascertain test execution times. This data is crucial for optimizing subsequent test runs. In the third step, we initiate the test item collection, which is executed through the invocation of `pytest`, configured with specific command-line options tailored to our testing requirements. Upon collection, the fourth step entails assigning the previously determined test execution times to the respective test items, wherever applicable. Step five involves inputting these test items, now augmented with their expected execution times, into our test subset solver. This solver is instrumental in determining the optimal workload distribution across each worker node, thereby enhancing overall testing efficiency. The sixth step marks the execution phase, where the test sets, as determined in the previous step, are run on the respective worker nodes. Finally, in the seventh and concluding step, we systematically release the DUTs that were initially reserved, thereby completing the testing cycle and preparing the infrastructure for future test runs.

### D. Synchronization Mechanisms

Tests executed in parallel may compete for shared resources in their environment, such as networking server ports, files in a shared filesystem, database connections, or computing resources like CPU and disk I/O.

In order to coordinate the access to these resources, multiple synchronization mechanisms exist in general [13] and, relevant for our situation, in the Python standard library [7], such as (recursive) locks, events, conditions, and semaphores. The synchronization mechanisms provided by the Python standard

```

1 Inputs:
2   items: Dictionary of Python modules to ModuleTestItems
3   num_workers: Total number of workers for distribution
4   split_threshold: Threshold for distributing module tests across buckets
5
6 Outputs:
7   buckets: list of test buckets to be executed by individual workers
8
9 Initialize:
10  buckets = [new Bucket() for i in 1 to num_workers]
11
12 Helper Function calc_min_bucket_idx(buckets):
13   Return index of bucket with the lowest execution weight
14
15 Distribute Weighted Test Items:
16   For each module_items in items:
17     total_weight = sum of weights in module_items.weighted
18     If split_threshold is defined and total_weight < split_threshold:
19       min_idx = calc_min_bucket_idx(buckets)
20       buckets[min_idx].extend(module_items.weighted)
21     Else:
22       For each module_item in module_items.weighted:
23         min_idx = calc_min_bucket_idx(buckets)
24         buckets[min_idx].append(module_item)
25
26 Distribute Unweighted Test Items:
27   idx = 0
28   For each module_items in items:
29     If there exists an empty bucket:
30       empty_bucket = find first empty bucket in buckets
31       empty_bucket.extend(module_items.unweighted)
32     Else:
33       buckets[idx].extend(module_items.unweighted)
34       idx = (idx + 1) % num_workers
35
36 Return buckets

```

Listing 1. Simplified 1D Cutting Stock Algorithm

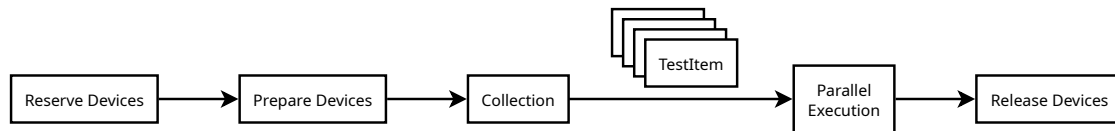


Fig. 3. The Big Picture: High-Level Test Run

library are appropriate primarily for scenarios where there is a parent-child relationship between processes executing Python interpreters. In our CI/CD pipelines, we concurrently run multiple test instances. While these tests may run on the same node within our test infrastructure, it is not a necessity, thus giving rise to the requirement for synchronization mechanisms that operate effectively across node boundaries. To facilitate locking that spans different systems, the implementation of distributed locks becomes essential. Nevertheless, many existing methods for distributed locking depend on a central component, such as a broker or coordinator, which introduces additional complexity into our test execution architecture. For environments based on Python, Redis offers robust support for implementing distributed lock patterns, as detailed in [1]. Another viable alternative is Apache ZooKeeper [8], which is accessible via the `Kazoo` Python library, among other tools, as cited in [3].

Performance issues can arise when locking access to shared

resources in a test environment. A viable solution to mitigate this is by duplicating the shared resource. For instance, in our testing setup, we allocated a unique range of server ports to each Device Under Test (DUT). This strategy enabled simultaneous opening and closing of server ports required by different test cases, effectively preventing interference across tests involving various DUTs. Nevertheless, this approach necessitates the capability of both client and server services to accommodate dynamic port allocation. It ensures that they can operate flexibly with varying ports, which is essential for the successful implementation of this solution.

### E. Speedup of Parallel Test Execution

In our initial experiments, we focused on a comprehensive test suite comprising roughly 1,600 tests. Each test was characterized by complex interdependencies involving both hardware and software components. These dependencies varied,

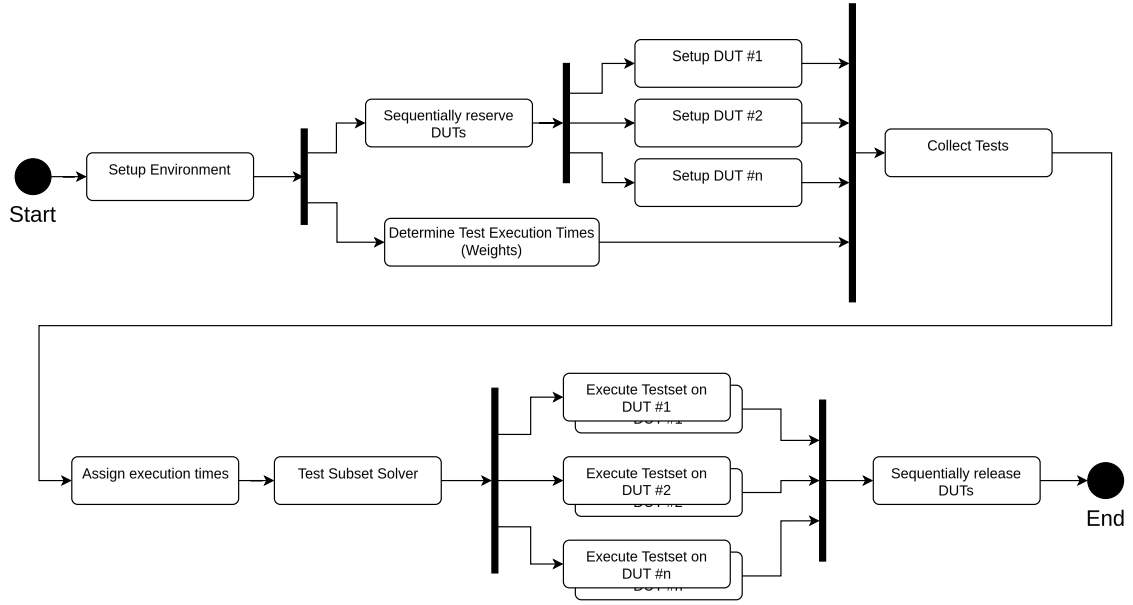


Fig. 4. Parallelization of the Test Execution Cycle

encompassing tasks such as device reboots as mandated by test specifications, as well as the setup and teardown of service infrastructures like server processes, and returning devices to their original state.

When executed sequentially, this suite required 4 hours and 24 minutes to complete. However, the application of our proposed methodology, even without the integration of estimated execution durations (weights) for the 1DCSP, significantly reduced the execution time to 35 minutes when using 20 DUTs. This reduction translates to an approximate speedup factor of 7.5 times. Due to constraints in experimenting with this extensive test setup, we conducted additional tests on a smaller scale, consisting of approximately 110 test cases. In this reduced scenario, the average execution time with only one Device Under Test (DUT) was about 23 minutes and 30 seconds, with a standard deviation of 85 seconds. To further optimize the distribution of tests across all available DUTs, we assigned estimated execution duration markers to all tests exceeding 1 second. This approach aimed to leverage our distribution strategy more effectively.

Figure 5 depicts the total test execution duration in dependence to the number of DUTs used and Figure 6 shows the correlation between the number of Devices Under Test (DUTs) and speedup factor when comparing to sequential test execution, i.e. using only one DUT. For instance, employing 4 DUTs resulted in a 3.36x speed increase, which further escalated to 5.22x with 8 DUTs.

When we expand the DUT count beyond 8, the importance of optimizing and modularly separating test cases becomes more pronounced, since the overall test run time is constrained by the duration of the longest test case. This leads to a 'speedup ceiling,' where speedup factors begin to plateau. This plateau is influenced by both the number of DUTs and the level of test op-

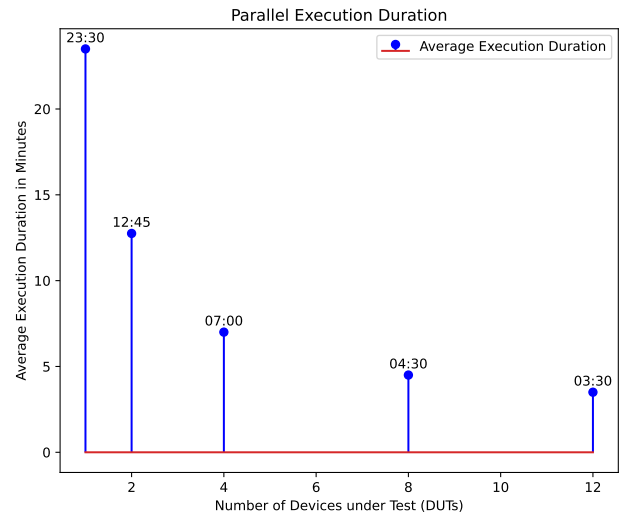


Fig. 5. Parallel Execution Duration

timization. For instance, increasing the DUT count from 1 to 6 (in theory) yields an approximately 5x speed improvement, but doubling it further to 12 only adds less than 2x increase. This illustrates how the duration of the longest test case becomes a critical factor in determining overall efficiency, underscoring the need for effective test optimization to maximize throughput. In our specific scenario, with the longest running test taking 2 minutes and 50 seconds, the calculated ceiling is:

$$\text{Speedup Ceiling} = \frac{23 \times 60 + 30}{2 \times 60 + 50} \approx 8.30. \quad (1)$$

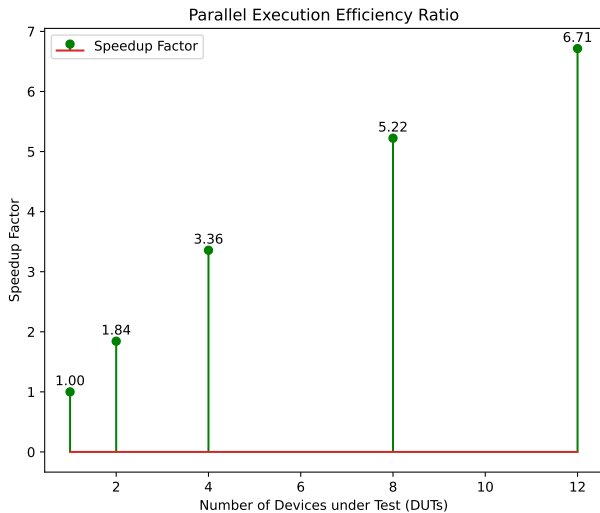


Fig. 6. Parallel Execution Efficiency Ratio

## CONCLUSION AND OUTLOOK

In conclusion, our study demonstrates that parallel execution of software tests yields significant benefits. The enhancement in test execution performance can be attributed to the strategic utilization of historical test execution data. This data plays a crucial role in the determination and assignment of execution weights, a key component of our adapted cutting-stock algorithm. Our research highlights the invaluable impact of historical test data in refining testing strategies, leading to more streamlined and effective methodologies in software development. This approach not only accelerates the testing process but also contributes to the development of more robust and reliable software products.

Looking ahead, our future research endeavors will focus on developing a dynamic feedback loop. This loop will continuously incorporate data from previous test executions to refine and adjust the weights used by the cutting-stock algorithm. This innovation aims to eliminate the manual assignment of test item weights, while ensuring that the weights accurately reflect real-world execution times. Additionally, we envision exploring advanced heuristics capable of identifying and potentially skipping tests unlikely to fail, as suggested by recent studies. Such enhancements promise to further optimize testing efficiency, reducing redundant test executions and sharpening the focus on critical test scenarios.

## REFERENCES

- [1] Distributed Locks with Redis. <https://redis.io/docs/manual/patterns/distributed-locks/>. Accessed: 2024-02-11.
- [2] JUnit 5 User Guide - Parallel Execution. <https://junit.org/junit5/docs/5.3.0-M1/user-guide/index.html#writing-tests-parallel-execution>. Accessed: 2024-01-29.
- [3] Kazoo. <https://github.com/python-zk/kazoo>. Accessed: 2024-02-11.
- [4] pytest: helps you write better programs. <https://pytest.org>. Accessed: 2024-01-26.

- [5] pytest-parallel. <https://github.com/kevlened/pytest-parallel>. Accessed: 2024-01-26.
- [6] pytest-xdist. <https://github.com/pytest-dev/pytest-xdist>. Accessed: 2024-03-06.
- [7] Python Multiprocessing - Synchronization primitives. <https://docs.python.org/3/library/multiprocessing.html#synchronization-primitives>. Accessed: 2024-02-11.
- [8] Welcome to Apache Zookeeper™. <https://zookeeper.apache.org/>. Accessed: 2024-02-11.
- [9] Welcome to labgrid. <https://labgrid.org>. Accessed: 2024-01-26.
- [10] M Bertolini, D Mezzogori, and F Zammori. Hybrid heuristic for the one-dimensional cutting stock problem with usable leftovers and additional operating constraints. *International Journal of Industrial Engineering Computations*, 15(1):149–170, 2024.
- [11] Victor Cheruiyot. Machine learning driven software test case selection. Master's thesis, Concordia University of Edmonton, 2021.
- [12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 4 edition, 2022.
- [13] Edsger W Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 65–138. Springer, 2002.
- [14] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 2003.
- [15] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The art of testing less without sacrificing quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 483–493. IEEE, 2015.
- [16] Xianhao Jin and Francisco Servant. What helped, and what did not? an evaluation of the strategies to improve continuous integration. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 213–225. IEEE, 2021.
- [17] Timothy G. Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Pearson Education, 2004.
- [18] Armin Najafi, Weiyi Shang, and Peter C Rigby. Improving test effectiveness using test executions history: An industrial experience report. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 213–222. IEEE, 2019.

## AUTHORS AND AFFILIATIONS

**Rainer Poisel** a founder at honeytreeLabs, is known for his broad expertise in software development and architecture, coupled with effective team leadership. He excels in integrating complex systems and tackling modern IT challenges through his deep technical insight and interdisciplinary approach. His work, focusing on practical applications and exploring new technologies are key drivers behind honeytreeLabs' reputation for delivering innovative IT services and solutions.

**Stefan Riegler** a founder at honeytreeLabs, is distinguished by his extensive experience in IT and embedded security. He is known for his hands-on, all-rounder approach and focuses on areas such as cyber security, forensics, reverse engineering, and radio frequency protocols. His role at honeytreeLabs involves creating customized IT solutions, with a particular emphasis on cyber security.

**honeytreeLabs Cooperation** offers specialized services in software development and cybersecurity, focusing on digitization and automation. We provide comprehensive IT solutions, including consulting and project management, to enhance business efficiency and cybersecurity. Dedicated to technical innovation, honeytreeLabs aims to bolster business competitiveness with tailored IT strategies.