COSMOS
DevOps for Complex Cyber-physical Systems

Project Number 957254

D3.6 Approach for the smart allocation of jobs on HiL and simulators, and build prioritisation

Version 1.0
22 December 2021
Final

Public Distribution

University of Sannio and Intelligentia

Project Partners: **Aicas, Delft University of Technology, GMV Skysoft, Intelligentia, Q-media, Siemens, Siemens Healthcare, The Open Group, University of Luxembourg, University of Sannio, Unparallel Innovation, Zurich University of Applied Sciences**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the COSMOS Project Partners accept no liability for any error or omission in the same.

© 2021 Copyright in this document remains vested in the COSMOS Project Partners.

Project Partner Contact Information

Aicas James Hunt Emmy-Noether-Strasse 9 76131 Karlsruhe Germany Tel: +49 721 663 968 0 E-mail: jjh@aicas.com	**Delft University of Technology** Annibale Panichella Van Mourik Broekmanweg 6 2628 XE Delft Netherlands Tel: +31 15 27 89306 E-mail: a.panichella@tudelft.nl
Intelligentia Davide De Pasquale Via Del Pomerio 7 82100 Benevento Italy Tel: +39 0824 1774728 E-mail: davide.depasquale@intelligentia.it	**GMV Skysoft** José Neves Alameda dos Oceanos Nº 115 1990-392 Lisbon Portugal Tel. +351 21 382 93 66 E-mail: jose.neves@gmv.com
Q-media Petr Novobilsky Pocernicka 272/96 108 00 Prague Czech Republic Tel: +420 296 411 480 E-mail: pno@qma.cz	**Siemens** Birthe Boehm Guenther-Scharowsky-Strasse 1 91058 Erlangen Germany Tel: +49 9131 70 E-mail: birthe.boehm@siemens.com
Siemens Healthineers David Malgiaritta Siemensstrasse 3 91301 Forchheim Germany Tel: +49 9191 180 E-mail: david.malgiaritta@siemens-healthineers.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Sannio Massimiliano Di Penta Palazzo ex Poste, Via Traiano I-82100 Benevento Italy Tel: +39 0824 305536 E-mail: dipenta@unisannio.it	**University of Luxembourg** Domenico Bianculli 29 Avenue J. F. Kennedy L-1855 Luxembourg Luxembourg Tel: +352 46 66 44 5328 E-mail: domenico.bianculli@uni.lu
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	**Zurich University of Applied Sciences** Sebastiano Panichella Gertrudstrasse 15 8401 Winterthur Switzerland Tel: +41 58 934 41 56 E-mail: panc@zhaw.ch

Document Control

Version	Status	Date
0.1	Document outline	12 October 2021
0.2	Introduction and Design draft	5 November 2021
0.3	Results draft	26 November 2021
0.4	First full draft	12 December 2021
0.5	Further editing draft	15 December 2021
0.8	QA review	21 December 2021
1.0	Final Version	22 December 2021

Table of Contents

Executive Summary

One of the challenges faced by practitioners when setting and evolving a CI/CD pipeline for CPS development is to cost-effectively use the limited number of simulators (e.g., Carla and BeamNG) and hardware devices resources. To deal with the above challenges, it is important to optimize the resources' usage by properly allocating tasks/jobs across simulators and/or HiL used in the pipeline.

This deliverable proposes an approach (i.e. Build Allocation) aimed at optimizing the allocation of simulators and HiL taking into account dependencies between different tasks/jobs. Specifically, three different aspects are considered by the proposed approach when searching for the optimal allocation: (i) minimize the overall build execution time, (ii) maximize the test effectiveness, (iii) proper usage of simulators and hardware devices, i.e. guarantee a uniform allocation above different devices, trying to minimize the cost in the presence of third-parties resources.

Build Allocation provides a set of "optimal solutions", each one allocating tasks/jobs on the available resources. It takes as inputs (i) the execution time belonging to each task, (ii) the priorities and dependencies across different tasks, (iii) the time budget provided by the developers in terms of upper bound for the overall build execution time, as well as (iv) the number of available resources (i.e. simulators and real devices). In order to do so, Build Allocation uses multi-objective optimization relying on genetic algorithms.

Build Allocation has been preliminarily evaluated in two different contexts: (i) a real case study with a test suite of of 72 test cases belonging to a satellite on-board software system, and (ii) synthetic data. Test case data are shared from one company of the consortium: Intelligentia.

Furthermore, the approach has been compared with a baseline (i.e. a random allocation approach), showing promising results. Indeed, Build Allocation outperforms the random allocation in both contexts.

As future work we plan to compare our approach also with greedy algorithms, e.g. integer programming. Details about implementation and integration will be discussed in D3.7 deliverable.

1 Introduction

The development of Cyber-Physical Systems (CPSs) requires the use of either simulators or Hardware-in-the-Loop (HiL) during various stages of the development activity, and in particular upon testing the systems [2, 22, 11]. For various reasons, such resources could be scarce for developers:

1. HiL is only available on the customer's premises. Imagine, for example, a company developing a software for a train or an airplane;
2. In some cases, even the simulator is unavailable on the developer's or tester premises, e.g., when the customer does not want to disclose the simulator, or in scenarios in which a third-party is acting as validator of somebody else's software;
3. Even when HiL or simulators are available on the developer/tester environment, they constitute scarce resources and therefore they must be used with parsimony.

In addition to what said above, CPSs builds can be very expensive to run, for various reasons

1. slow deployment on HiL.
2. large test suites to be executed into a given time budget;
3. tests cases have to be executed in conjunction with simulators/HiL not always available;
4. expensive static analysis;
5. expensive test cases;

To cope with the aforementioned issues, it is necessary to perform two kinds of actions. First, given an incoming build, it may or may not be necessary to execute it immediately [23, 9]. That is, if a build helps a developer to discover faults related to the module she just completed, and if the build execution would be fast, then it may be worthwhile to execute it. Otherwise, the build jobs could be rescheduled in a nightly or in general in a periodic build.

Second, if multiple resources (e.g., HiL or simulators) are available, a monolithic build could be decomposed into multiple, parallel ones. For example, if a build contains several tests, the tests could spread over the available machines (HiL/simulators) with the goal of minimizing the overall build time.

Third, if a given time budget is available, it would be wise to select the tests (or static analyses) with the highest priority, decompose them into multiple jobs, and allocate those jobs onto multiple servers, so to maximize the test/static analysis utility while keeping the build completion time within the budget, e.g., the 10 minutes suggested by Duvall for continuous builds [9].

The goal of this deliverable is to propose an approach to deal with the last two points mentioned above, whereas the first one, i.e., determining the build outcome and allocating build jobs towards continuous or nightly builds, will be the goal of D3.7.

More specifically, we propose a multi-objective search-based approach to allocate build jobs (or, in the simplest scenario, test cases) onto multiple servers. The approach is based on the Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [7] which produces Pareto-optimal sets of solutions that, given the availability of a given number of HiL/simulators, are aimed at minimizing the overall completion time and maximizing the test/analysis utility (e.g., expressed in terms of priority, or test coverage). Also, the algorithm allows for specifying constraints in terms of maximum allowed time budget, as well as dependencies between jobs.

The delivery describes the approach and reports preliminary evaluation results on data from an industrial project and synthetic data generated from the real ones.

2 Background

Allocating resources to a software project or make complex and difficult decisions, like assigning tasks to teams, constitute crucial activities that affect project cost and completion time [15, 8]. Finding a solution to these problems is intrinsically NP-hard, and exhaustive approaches are not viable due to the size and complexity of many software projects. Therefore, during recent years, several software-related problems have been formulated as optimization problems and resolved with meta-heuristics. The idea of solving software engineering problems using search-based optimization techniques has been named "Search-Based Software Engineering" [5].

This section overviews on search-based optimization techniques, like Genetic Algorithms and Multi-Objective optimization. We also analyze the state of the art regarding build outcome prediction and build test optimization. We also want to make an overview on the state of the art related build optimization.

2.1 Preliminaries

In the following we recall some preliminaries on Genetic Algorithms and Multi-Objective Optimization.

2.1.1 Genetic Algorithms

Genetic Algorithms (GAs) [12] belong to the family of evolutionary algorithms that, inspired by the theory of natural evolution, simulate the evolution of species emphasizing the law of survival of the strongest to solve, or approximately solve, optimization problems. Thus, these algorithms create consecutive populations of individuals, considered as feasible solutions for a given problem (phenotype) to search for a solution which gives the best approximation of the optimum for the problem under investigation. To this end, a fitness function is used to evaluate the goodness (i.e., fitness) of the solutions represented by the individuals, and genetic operators based on selection and reproduction are employed to create new populations (i.e., generations).

Algorithm 1: Pseudo-code of a Genetic Algorithm.

```
1 t ← 0;
2 initialize a population P(t) of n individuals
3 while termination-condition not met do
4 |   t ← t + 1;
5 |   select a subset P'(t − 1) of individuals from P(t − 1) to reproduce
6 |   apply crossover to P'(t − 1) and introduce offspring in P(t)
7 |   mutate individuals in P(t)
8 |   evaluate P(t)
9 end
```

As shown in Algorithm 1, the elementary evolutionary process of these algorithms is composed of the following steps:

1. a random initial population P(0) is generated (line 1) and a fitness function is used to assign a fitness value to each individual (line 2);

2. given t the current generation (line3),some individuals of a population P'(t-1) are selected as parents (line 6) and new individuals are created by applying genetic operators (i.e., crossover and mutation). The crossover operator (line 7) combines two individuals (i.e., parents) to form one or two new individuals

(i.e., offspring), while the mutation operator (line 8) is used to randomly modify an individual. Then, to determine the individual that will survive among the offspring and their parents a survivor selection is applied according to the individuals' fitness values (line 9);

3. step 2 is repeated until stopping criteria hold.

When designing a GA, the crossover and mutation operators play a crucial role. Different crossover operators can be used. Among the most used, there are:

- One-point crossover. A point in the chromosome of the two parents is selected, and all the genes beyond that point in either chromosome are swapped between the two parents.
- Two-point crossover. Two points in the chromosome of the two parents are selected, and everything between the two points is swapped between the parents, generating the offspring.
- Uniform crossover. A fixed mixing ratio between two parents is used. Unlike one- and two-point crossover, the uniform crossover enables the parent chromosomes to contribute the gene level rather than the segment level.

As for the mutation, the selection of the operator depends on the representation of the solution. For integer and float genes, a widely-used operator is the uniform mutation. Using such an operator, the value of a chosen gene is replaced with a uniform random value selected between the user-specified upper and lower bounds for that gene. If the solution is represented by a binary string a common mutation operator is the bit flip, where the bit of the chosen gene is inverted (i.e., if the value is 1, it is changed to 0 and vice versa).

It is worth noting that during each generation these operators are applied with a certain probability, named crossover rate and mutation rate. In addition, at each generation parents have to be selected for crossover and mutation. Thus, also the selection operator plays an important role.

The most used selectors are the roulette wheel and the tournament selection. In the former, each individual's probability of selection is directly proportional to its relative fitness. In the latter, small subsets of the population are selected randomly (a tournament) and the fittest member of the subset is selected for the next generation.

Finally, the stopping criterion for the evolutionary process is usually based on a maximum number of generations. This stopping criterion can be combined with other criteria to reduce the computation time. For example, the search process can be stopped when there is no improvement in the fitness value for a given number of generations.

A variant of GAs is Genetic Programming (GP) [14], where the aim is to generate programs (that can be also prediction models, or expressions, etc.) having certain properties. The representation is often (but not necessary) a program Abstract Syntax Tree (AST) and the fitness is evaluated by executing the program.

2.1.2 Multi-Objective Optimization

An optimization problem can have one objective, but also more than one objective (multi-objective optimization). In a multi-objective optimization problem, a solution is described in terms of a decision vector (x1,x2,...,xn) in the decision space X. Then the fitness function f : X -> Y evaluates the quality of a specific solution by assigning it an objective vector (y1,y2,...,yk) in the objective space Y, where k is the number of objectives.

Comparing solutions in multi-objective optimization is not as trivial as in the case of single-objective optimization problems. Specifically, in multi-objective optimization problems it is necessary to exploit the concept of Pareto dominance: an objective vector y1 is said to dominate another objective vector y2 (y1 > y2) if no component of y1 is smaller than the corresponding component of y2 and at least one component is greater. The Pareto dominance allows to say that a solution x1 is better than another solution x2, i.e., x1 dominates x2 (x1 >

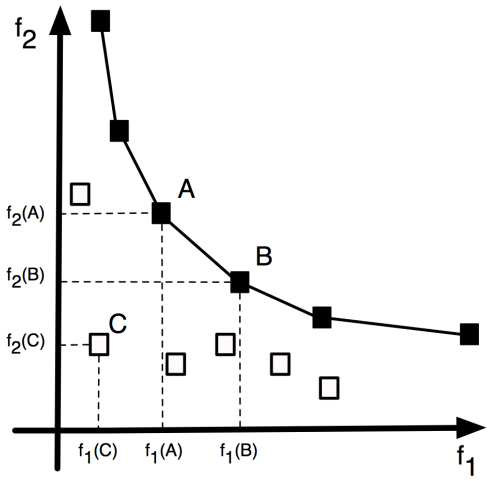


Figure 1: Pareto.

x2), if f(x1) dominates f(x2). For example, in Figure 1, point C is dominated by A and B since f1(A) > f1(C), f2(A) > f2(C), f1(B) > f1(C), and f2(B) > f2(C). Instead, A and B represent non- dominating solutions: if we consider A, there is at least another solution (B in our case) such that f1(B) > f1(A). Similarly, if we consider B, there is at least another solution (A) such that f2(A) > f2(B). It is worth noting that using such a definition it is possible to define a set of optimal solutions, i.e., solutions not dominated by any other solution. Such solutions may be mapped to different objective vectors. In other words, there may exist several optimal objective vectors representing different trade-offs between the objectives. This set of optimal solutions is generally denoted as the Pareto set $X^* \subseteq X$, while the fitness values achieved by such solutions represent the Pareto front $Y^* \subseteq Y$.

In principle, a multi-objective optimization problem can be reduced to a single- objective optimization problem. For instance, the different objectives can be aggregated into a single one. However, the analysis of the Pareto front can help the decision maker in (i) selecting the most suitable solution, i.e., the solution that provides the best compromise in a particular scenario; and (ii) analyzing the trade-off provided by each solution.

The concept of Pareto dominance is also used to rank solutions and to apply selection strategies based on non-domination ranks. Generally, such algorithms are elitist: the best solutions—i.e., the non-dominated solutions—are either kept in the population itself or are stored separately to be reused. In the first case, they participate to the reproduction process. However, the number of non-dominated solutions might greatly increase with the number of objectives, which limits the number of places reserved for new individuals. Therefore, such algorithms generally use a specific operator to preserve diversity. The elitist Non-dominated Sorting Genetic Algorithm (NSGA-II) [7] is certainly one of the most popular algorithms belonging to this category. A naive multi-objective optimization algorithm would require O(M N) comparisons to identify each solution of the first non-dominated front in a population of size N and with M objectives, and a total of $O(MN^2)$ comparisons to build first non- dominated front. This is because each solution needs to be compared with all other solutions. Since the above step has to be repeated for all possible fronts—which can be at most N, if each front is composed of one solution—the overall complexity for building all fronts is $O(MN^3)$.

NSGA-II uses a faster algorithm for non-dominated sorting, which has a complexity $O(MN^2)$:

1. for each solution p in the population, the algorithm finds the set of solutions Sp dominated by p and the number of solutions np that dominate p. The set of solutions with np = 0 are placed in the set first front F1.

2. $\forall p \in F_1$, solutions $q \in S_p$ are visited and, if $n_q - 1 = 0$, then solution q is placed in the second front F_2. This step is repeated $\forall p \in F_1$ to generate F_3, etc.

To compare solutions within the same non-dominated front, NSGA-II uses the "crowded comparison operator". That is, given two solutions x_1 and x_2, x_1 is preferred over x_2 if it belongs to a different (better) front. Otherwise, if x_1 and x_2 belong to the same front, the solution located in the less crowded region of the front is preferred. Then, NSGA-II produces the generation $t + 1$ from generation t as follows:

1. generating the child population Q_t from the parent population P_t using the binary tournament selection and the crossover and mutation operators defined for the specific problem;
2. creating a set of 2N solutions $R_t \equiv P_t \cup Q_t$;
3. sorting R_t using the non-domination mechanism described above, and forming the new population P_t+1 by selecting the N best solutions using using both the dominance ranking and the crowding distance.

2.2 State of the Art

In the context of Continuous Integration, previous works focus on optimizing mainly test case execution looking to cost, coverage, and fault detection, or their combinations. Machine learning algorithms and multiple heuristic techniques are used to prioritize test cases.

Another challenging topic for software engineering research is analyzing continuous integration build failures. Many works use prediction models to predict the CI build status. The main aim is to drop the cost of CI.

In this section, we make an overview of current literature mainly focusing on build optimization and also on build outcome prediction.

2.2.1 Build Optimization: Test Cases Prioritization

Marijan et al. [17] present an approach for Prioritization for Continuous Regression Testing, named ROCKET. They use a weighted function to compute test priority. Prioritization is based on historical failure data, test execution time, and domain-specific heuristics. The weights are higher if tests uncover regression faults in recent iterations of software testing and reduce time to detection of faults

Elbaum et al. [10] examine the use of Regression Test Selection (RTS) and Test Case Prioritization (TCP) techniques to increase the cost-effectiveness in CI. The proposed approach aims to improve regression testing in CI development. They use time windows, analyzing the history, to determine which tests have to be executed with higher priority and what tests are worth executing. Finally, they evaluate the proposed approach, through an empirical study on a Google dataset.

Alazzam and Nahar [3] introduce a new algorithm that aims to prioritize test cases in the test suite starting from method and line of code coverage. Test cases which cover the most methods and lines of code are more effective and efficient in finding errors. The algorithm combines two coverage metrics: the percentage of the covered method along with the percentage of the covered line of code.

Spieker et al. [20] define a new method (named Retecs) to select and prioritize test cases in CI. Relics through reinforcement learning method can learn information, i.e. test cases duration, last execution, and failure history. Observing historical information and previous CI cycles the method can prioritize higher error-prone test cases. Three industrial case studies are used to evaluate the proposed method.

Also, Wu et al. [21] use reinforcement learning to select and prioritize test cases but differently from [20] they apply a time window-based approach to extract more valuable historical information from recent test cases to earlier ones. They also investigate the most suitable window size starting from the impact of size over fault detection capability and time cost. Wu et al. evaluate their approach on three industrial CI datasets.

Liang et al. [16] propose a novel approach for continuous prioritization of commits. For each arrival of a new commit, the proposed algorithm creates a commit queue, and then using test suite failure and execution history

information it prioritizes the commit executions in the queue. Thus, the proposed approach can prioritize commits concerning each previously scheduled commit and also to the arrival of a new commit. Three non-trivial CI data sets are used to evaluate the proposed approach.

Abdalkareem et al. [1] use machine learning techniques to determine which commits unnecessarily kicking off the CI process can be skipped. They analyze extract 23 commit-level features from 10 open-source Java projects using Travis CI. A decision tree is built to determine whether a commit can be a CI skip commit or not.

Differently from the above works, we plan to optimize not only test case execution but also improve the task allocation over different servers/simulators. We want to use Multi-objective Genetic Algorithms to achieve this goal. Using these techniques we can optimize tasks allocation on a pipeline of continuous integration taking into account multiple constraints, like time-budget constraints or task dependencies.

2.2.2 Build Outcome Prediction

Pan et al. [18] predict continuous test suite failure to reduce the cost of continuous integration. They present a machine learning-based approach able to predict whether a particular code change should trigger the test suite at all or not. Features extracted are related to code changes, the test suite, and the development history. They evaluate their approach over 15k test suite runs from 242 open-source projects.

Saidani et al. [19] use a novel search-based approach based on Multi-Objective Genetic Programming (MOGP) to predict the CI build outcome. They aim at finding the best combination of CI-built features and their appropriate threshold values, based on two conflicting objective functions to deal with both failed and passed builds. A benchmark of 56,019 builds from 10 large-scale and long-lived software projects that use the Travis CI build system is used to evaluate the proposed approach.

B. Chen et al. [4] conduct an empirical study on 2,590,917 builds to characterize build times in real-world projects, and a survey with 75 developers to understand their perceptions about build outcome prediction. They propose a new history-aware approach, named BuildFast, to predict CI build outcomes cost-efficiently and practically. They develop multiple failure-specific features from closely related historical builds via analyzing build logs and changed files, and propose an adaptive prediction model to switch between two models based on the build outcome of the previous build. 20 open-source projects are used to demonstrate the effectiveness and efficiency of BuildFast.

Hassan et al. [13] propose a build prediction model that uses TravisTorrent data set with build error log clustering and AST level code change modification data. They predict whether a build will be successful or not without attempting actual build so that developers can get early build outcome results. Their prediction model is based on combined features of the build-instance metadata and code difference information of the commit.

Conversely, our approach aims at optimizing tasks allocation in the context of CI. As future work, we plan to investigate for build outcome prediction. For now, our algorithm takes into account the build outcome as an input parameter. To know a predicted build outcome in advance allows to postpone successful build and prioritize failing build.

Note that in the context of CPS, to predict build outcome is challenging. For example, a build outcome can be associated to the output of a test case. Test cases executions depend on simulation data or values from physical sensors in the CPS context. Thus, it is challenging to predict tests outcomes. This challenge comes from flakiness due to simulators and HiL.

3 Proposed Approach

In this section we present the proposed approach for allocating jobs on multiple servers/simulators.

3.1 The Problem

When the team of developers CI/CD commits a new code change into a version control system (e.g. git, svn), the pipeline is triggered and the build process starts. Usually, a build process is composed of several jobs running in parallel. Each job collects different tasks generally executed in sequence. The most diffused tasks are Static Code Analysis, Build and Test. The continuous integration process aims at ensuring that enough testing is performed before code submission to avoid breaking builds and delaying the fast feedback that makes continuous integration desirable.

However, in the context of CPS, the task execution may require the usage of simulators and hardware-in-the-loop (HiL) which may have limited availability and may be shared across several projects. Thus, the number of jobs/tasks can be reduced and can be executed only jobs/tasks providing useful information. The problem is the optimal allocation of simulators and hardware with jobs/tasks prioritization.

The challenge is to allocate tasks cost-effectively and ensure a sustainable usage of the pipeline. For example, the build can be restructured in such a way that it performs tests concurrently on different sub-modules of a system. Test cases can be focused on a specific portion of code and all tests can be executed in parallel.

Our approach aims at optimizing the allocation of simulators and HiL with jobs/tasks prioritization. To achieve this goal we want to use multi-objective genetic algorithms. We want to prioritize jobs/tasks starting with their execution times and priorities taking into account jobs/tasks dependencies.

The ultimate goal is (i) minimizing the build time, (ii) maximizing the testing effectiveness, (iii) optimizing the usage of simulators and hardware devices, i.e., ensuring a uniform allocation and, if resources are rented from 3rd-parties, minimizing the costs.

3.2 Workflow of Proposed Approach

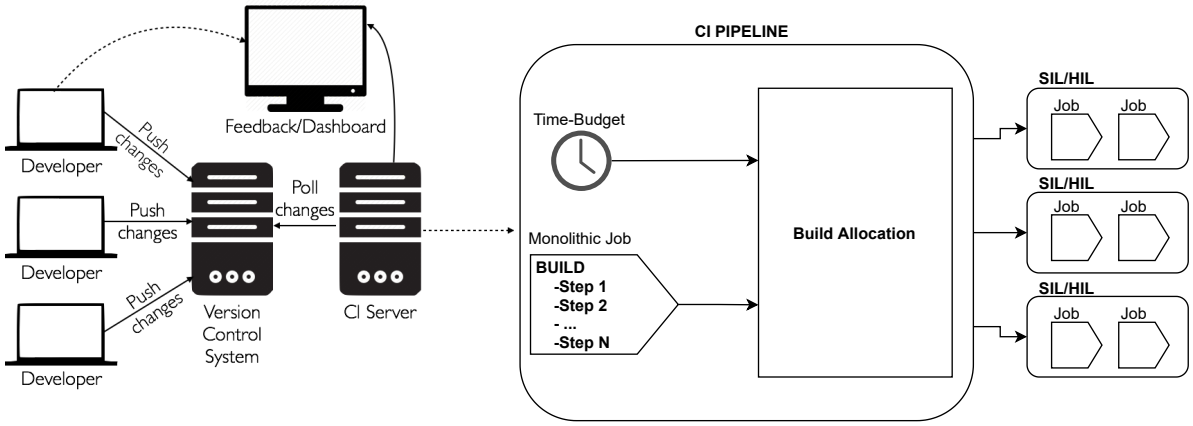


Figure 2: Build Allocation in CI Pipeline.

Figure 2 shows where the proposed approach fits into a CI pipeline. It is named Build Allocation and takes as input task execution times, task priorities, their dependencies, and a time budget to compute the optimal allocation of tasks on simulators and HiL. Our approach is triggered by the arrival of a new change into the version control system. The output is an optimal distribution of tasks/jobs between different simulators and HiL.

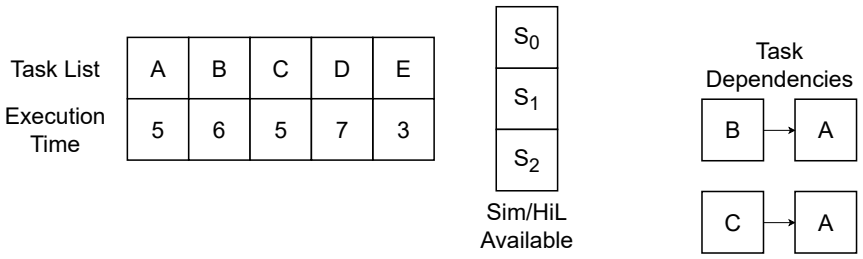
To optimize jobs/tasks allocation, our approach uses multi-objective genetic algorithms. Build Allocation carries out two formulations:

- the first one assigns each job to only one simulator/HiL;

- the second one takes into account also the possibility to duplicate jobs over different simulators/HiL to improve parallelism between jobs.

More precisely, the first formulation does not include the possibility to duplicate jobs/tasks between different simulators/HiL. Employing this formulation, dependent jobs/tasks have to be located on the same simulator/HiL. The second solution allows duplicating jobs/tasks. It takes into account the possibility to replicate the same job/task on different simulators/HiL. This duplication is useful in the case of dependent jobs/tasks since duplicating a job that has more than one dependent can improve the load balancing between all available simulators/HiL.

An example can be useful to better understand how these formulations work. Figure 3 presents an example scenario.



(a) Build Configuration Example.

(b) Testing of the Y-axis.

Figure 3: Example Scenario.

Figure 3a shows an example of build configuration. It considers a monolithic build job composed of 5 different tasks A, B, C, D, and E. To complete each task a precise amount of time is required. Each execution time is shown in Figure 3a under each tasks. The simulators/HiL available are 3: S_0, S_1 and S_2.

Dependencies between tasks are shown in Figure 3b. More precisely, B and C tasks depend on task A. It means that there is a correlation between these tasks: task B to be executed needs a preliminary execution of task A, the same is for task C. There is no dependency between B and C. Further D and E tasks are independent. Independent tasks can be executed independently of each other, also on different simulators/HiL.

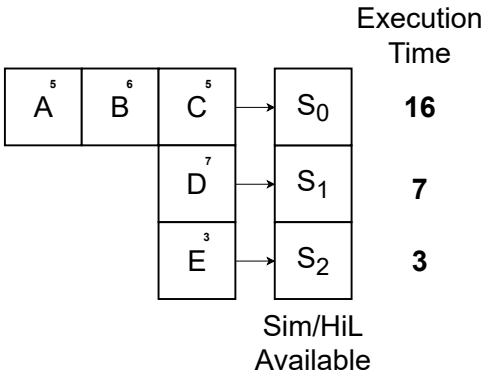


Figure 4: A possible optimal allocation using first solution proposed.

A possible outcome can be the one shown in Figure 4 by applying the first formulation. As Figure 4 shows, A, B and C tasks are located on S_0, D task is placed on S_1 and finally, E task is on S_2. If we look at the total execution time of every machine we can notice that the workload of S_0, i.e. 16, is greater than the other two times (7 and 3 respectively). It should be underlined that the total workload of each machine is computed as the sum of execution times of all allocated tasks.

This formulation does not allow duplicate jobs/tasks between several simulators/HiL: tasks dependent on each other have to be placed on the same machine. Using this approach the workload between different simulators/HiL maybe not be optimally balanced since dependent tasks have to be placed on the same simulator/HiL.

In this way, it may happen that there is a machine with a higher workload than the others, like the example in Figure 4 shows.

This is the reason why the proposed approach takes into account also a second formulation able to duplicate jobs over different simulators/HiL. Figure 5 shows a possible outcome.

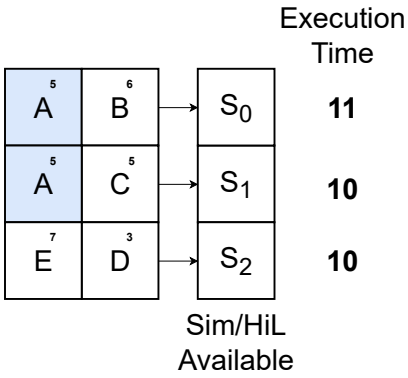


Figure 5: A possible optimal allocation using second solution proposed.

Figure 5 shows that B and C tasks now are split on S_0 and S_1 respectively. It is possible since task A is duplicated on the two simulators/HiL. If we look at the total workload of every simulator/HiL we notice that now it is better balanced compared with the previous solution.

3.2.1 Genetic Representation

Two genetic representations of individuals are defined to implement the two proposed formulations.

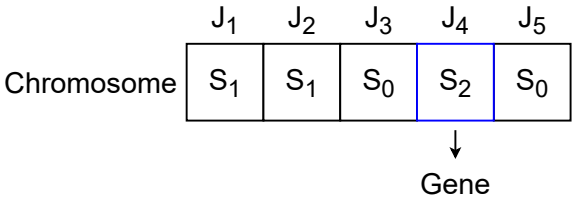


Figure 6: Individual of the first solution proposed.

Figure 6 shows an example of the defined individual for the first formulation to our jobs optimization problem. In this case, every individual is an array of integers. Each position of the array corresponds to a job that needs to be run on the pipeline. Thus, our chromosome is an array of jobs. In any position there is an integer representing the ID of a server/simulator where the job can be located, i.e, each gene is the server/simulator ID.

The size of the chromosome corresponds to the number of jobs that need to be allocated. The integer representing server/simulator ID is ranged between 0 and the number of available servers/simulators minus 1.

For example, considering our representation and Figure 6 we can affirm that, given 5 jobs and 3 servers/simulator, an individual can have the following allocation: 1 and 2 jobs on servers 1, 3, and 5 jobs assigned on server 0 and finally job 4 on server 1.

Figure 7 is an example of second formulation individual. In this case, the chromosome is an array containing integers. Each position of the array corresponds to an available server/simulator. Each gene is an integer corresponding to the binary representation of job allocation. More specifically, each gene is an array of binary numbers and, each position of this array corresponds to a job.

Thus, if we look at Figure 7, we can see that jobs 3 and 5 are allocated on server 2. This representation allows duplicating jobs since a job can be located on more than one server/simulator.

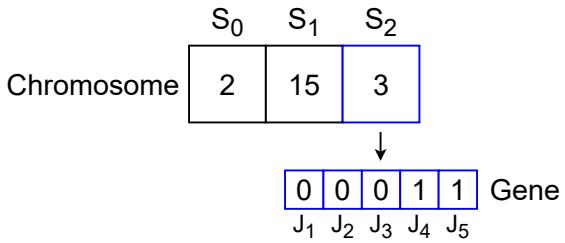


Figure 7: Individual of the second solution proposed.

3.2.2 Fitness Function and Constraints

To evaluate an individual of our population, we have defined a fitness function aiming to minimize the total execution time of jobs between different servers/simulators. To do so, we take into account the past execution time of any job. Given an individual, we compute for every server/simulator the sum of execution times of allocated jobs. At the end of this process, we know for every server/simulator their workload weight (expressed as total execution time). Our fitness function minimizes the maximum execution time. In this way, it tries to balance the workload between different machines.

Regarding constraints, we have two different types: the first one is related to the total workload of each server/simulator and the second one is related to dependencies between jobs. More in detail, we improve our solution limiting it to not exceed a given total workload and it has also to respect dependencies between jobs. Roughly speaking, we want that depending jobs are located on the same server/simulator, and also the total execution time of each server/simulator does not exceed a given amount of time.

3.2.3 Crossover, Mutation and Termination Criteria

In our solution, we use a half uniform crossover to generate new offspring. The half uniform crossover will first determine what indices are different in the first and the second parent. Then, it will take half of the difference to be selected from the other parent.

In the half uniform crossover scheme (HUX), exactly half of the non-matching bits are swapped. Thus first the Hamming distance (the number of differing bits) is calculated. This number is divided by two. The resulting number is how many of the bits that do not match between the two parents will be swapped.

The uniform Crossover method allows that each gene in the offspring is created by copying the corresponding gene from one or the other parent chosen according to a randomly generated binary crossover mask of the same length of the chromosomes. When there is a 1 in the crossover mask, the gene is copied from the first parent, and when there is a 0 the gene is copied from the second parent.

As mutation operator, we use polynomial mutation [6].

The termination criteria are fixed to 200 generations. To select a threshold for termination criteria we followed a step-by-step approach. We started number generation equal to 50 and then we proceeded step by step to increase it. We increase it until the number of solutions found was stable.

4 Preliminary Evaluation

In this section, we detail a preliminary evaluation of a real case study.

4.1 Case Study and Context

Our case study belongs to Intelligentia, a software company involved in verification and validation tasks for the aerospace domain. Thus, the scope of their pipeline is only for V&V since due to the safety integrity level of the CPS, the development and the V&V teams and pipelines must be kept distinct.

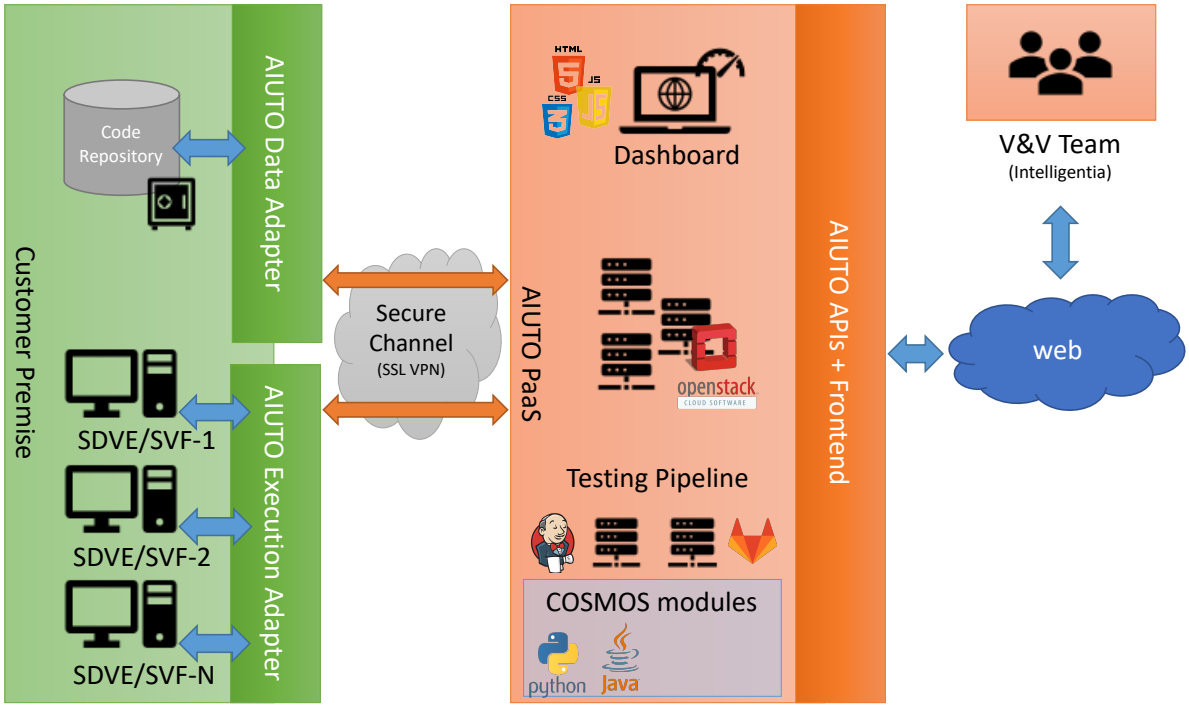


Figure 8: Intelligentia CI pipeline architecture.

Figure 8 shows the CI pipeline adopted by Intelligentia. Once a tester has written test cases, the pipeline pushes them on remote instances of a simulator where the software is built and tests executed.

Intelligentia team usually writes test cases starting from textual requirements and runs them relying on a limited number of third-party simulators. To preliminary evaluate the performance of our approach we use 72 test cases belonging to a satellite on-board software system that is going to be used on an ESA space mission. Test cases are written in C. On average, the execution time is equal to 116.2 sec.

The on-board software is made of various functionalities, called services, and these are divided into modules, called sub-services, that give to the software either the core functionalities or the advanced ones. In this specific case, during the functional testing of the on-board software, 72 test case have been identified and they refer to the following functional units:

- AOCS, orbit orientation management service;
- Utilities, group of utils of the on-board software such as buffer management, CPU load statistics, mathematical operations;
- EVAS, event action service used to manage the software reaction to software and hardware events;
- OBMS, an on-board monitoring system that monitors the on-board parameters;
- Forwarding Service, service that provides the functionality to send and receive packets among all the units connected to the on-board computer;
- Storage Service, service that manages the on-board data retrieval and archiving into physical disks;
- Log Service, service used to manage the system events log;
- Time Tools, specific utilities for the time management that is a critical component of a satellite system due to the fact that ground and spacetimes must be synchronized otherwise the on-board operation could not be executed at the exact time they should.

4.2 Data Extraction

To preliminary evaluate our approach we use 72 test cases for a satellite onboard software system. We extract the time execution and the related coverage starting from the final reports of execution. These reports are in HTML format. We use Python scripts to extract this information. To achieve this goal, we leverage the Python *BeautifulSoup* module. Such a module allows to pull data out of HTML and XML files. It provides idiomatic ways of navigating, searching, and modifying the parse tree.

4.3 Synthetic Data Generation

To assess our approach more extensively we generate also synthetic data. To produce distributions we fit

Starting from the distribution of 72 real test cases we generate three synthetic distributions of 500, 1000, and 2000 test cases respectively. Specifically, we use the time distribution of test cases to generate a new distribution of time including more samples. Fitting distributions to data is a very common task in statistics and consists in choosing a probability distribution modeling the random variable, as well as finding parameter estimates for that distribution. After the fitting, the real tests cases distribution is equivalent to a uniform distribution.

Thus, our synthetic dataset contains times randomly generated starting from the uniform distribution. More precisely, we adopt such distribution since the 72 test cases time distribution is identifiable as a uniform distribution.

4.4 Preliminary Results

The following sections detail the results obtained during the evaluation of Build Allocation algorithm on two different types of data: real and synthetic data. To compare the results achieved by Build Allocation we have considered as a comparison a random allocation.

4.4.1 Real Data Results: Intelligentia case study

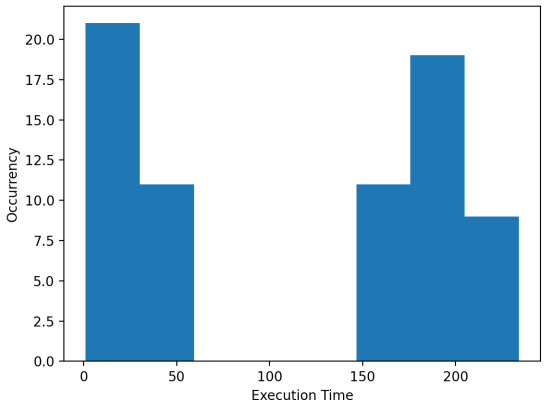


Figure 9: Tests Execution Time from real case study.

Figure 9 show how 72 test case times are distributed. The distribution is symmetric since it forms an approximate mirror image concerning the center of the distribution. The center of the distribution is easy to locate and both sides of the distribution are approximately the same length.

The distribution is comparable to a kind of uniform. The distribution has no modes or no value around which the observations are concentrated. Rather, we see that the observations are roughly uniformly distributed among the different values.

As the first test, we have evaluated Build Allocation algorithm using only one fitness function. In this single-objective optimization problem, there exists a single best solution that was found. The result directly contains the best-found values in the corresponding spaces. It aims at optimizing only the time distribution between different simulators/HiL. We compare the result achieved by Build Allocation with a random allocation.

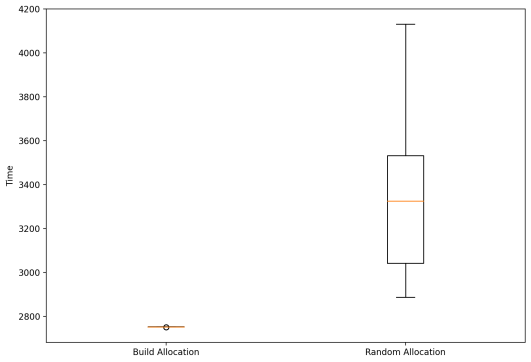


Figure 10: Build Allocation and Rand Allocation Comparison using 3 simulators/HiL.

For the first test, we use 3 simulators/HiL. We execute for 30 times both solutions: Build Allocation and random allocation.

Fig. 10 reports boxplots summarizing such results. It contains execution times distribution of two types of allocation, i.e. workload, of every simulator/HiL involved in this test.

The Build Allocation boxplot shows the execution times of optimal distribution of test cases between 3 simulators/HiL. The first and third quartiles are 2751 and 2754 respectively. The median execution time of each machine is 2753 seconds.

The second boxplot, named Random Allocation, shows the execution times distribution of random allocation of test cases between 3 simulators/HiL. In this case, the first and third quartiles are 3042 and 3533 respectively. The median execution time of each machine is 3326 seconds.

Comparing the two distributions of workload, we found a statistically significant difference between them. (Wilcoxon rank-sum test p-value < 0.05 and Cliff's Delta estimated large).

Results show that Build Allocation algorithms outperform the random allocation. Using an optimization function the allocation of test cases between several simulators/HiL is uniformly distributed in terms of time. All the machines involved during the execution of tests are correctly loaded balanced. Figure10 shows that using Build Allocation algorithm the total execution time on every simulator/HiL is stable.

Therefore the results show that optimal allocation leads to a more equitable balancing of the workload between several simulators/HiL.

To better evaluate Build Allocation approach we conduct another test aiming at minimizing the total execution time between 5 different simulators/HiL. We test if the optimizing algorithm is robust against a greater number of machines. Also in this test, we compare the results achieved by Build Allocation against a random allocation method.

Figure 11 shows the results obtained during this second test. We execute our Build Allocation algorithm and random solution 30 times. The Build Allocation boxplot shows the execution times of optimal distribution of test cases between 5 simulators/HiL. The first and third quartiles are 1653 and 1655 respectively. The median execution time of each machine is 1654 seconds.

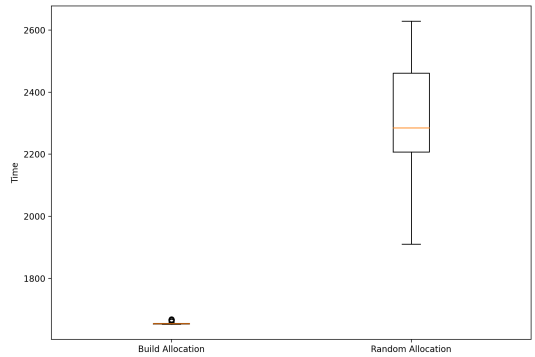


Figure 11: Build Allocation and Rand Allocation Comparison using 5 simulators/HiL..

The second boxplot, named Random Allocation, shows the execution times distribution of random allocation of test cases between 5 simulators/HiL. In this case, the first and third quartiles are 2208 and 2462 respectively. The median execution time of each machine is 2286 seconds.

We found a statistically significant difference between two distributions, Wilcoxon rank-sum test p-value $<$ 0.05 and Cliff's Delta estimated large.

Results in Figure 11 show that Build Allocation outperforms random solution. Also during this test, the optimal allocation of our approach correctly balances the workload between different simulators/HiL. Using random allocation the total execution time of each machine is not balanced. If we look at the second test we can easily see that the workload of S_1 is much smaller when compared to the load on other simulators/HiL.

Finally, we can say that the optimal solution computed using our approach outperforms the random allocation. It should be underlined that for this test we used only one optimization function aiming at minimizing the workload of each machine trying to correctly balance the total execution time of every simulator/HiL.

Thus, a further step to test our Build Allocation algorithm takes into account the multi-objective optimization. The following test includes two objective optimization functions. In this case, we take into account both time optimization and prioritization of test cases. We compare random allocation with Build Allocation algorithm using two fitness functions aiming at minimizing both the workload of every simulator/HiL and the priorities of every test.

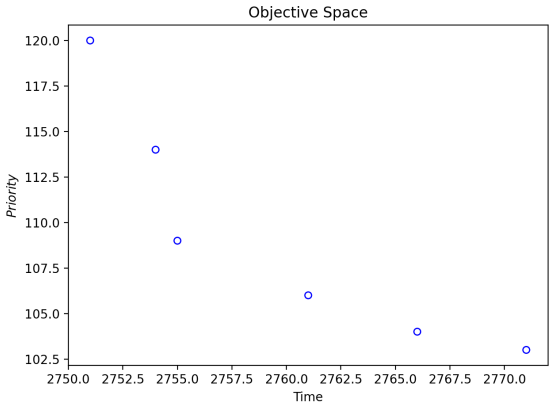


Figure 12: The Pareto-front obtained applying the Build Allocation optimization procedure.

Figure 12 shows the results after running of Build Allocation algorithm with two optimization functions. NSGA-II produced Pareto fronts composed of solutions for different staffing levels and different completion

times. In multi-objective optimization, the Pareto front (also called Pareto frontier or Pareto set) is the set of all Pareto efficient solutions. It shows the objective space of two objective functions. It is composed of 6 best solutions. Figure 12 shows a set of 6 non-dominated solutions. During this test, we involve 3 simulators/HiL.

To evaluate the results achieved we have used a random allocation. We execute our Build Allocation algorithm and random solution 30 times to compare obtained results.

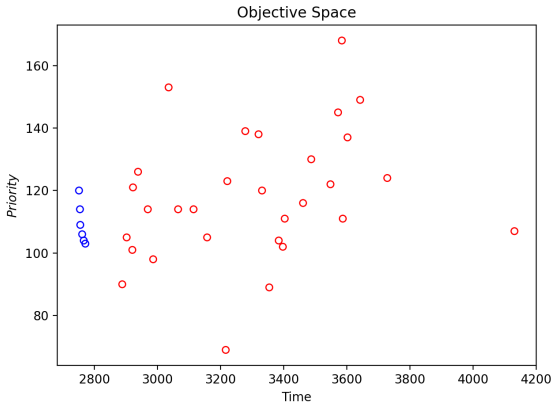


Figure 13: Build Allocation vs Random Allocation comparison results.

Figure 13 shows the results achieved by two different solutions. Solutions in blue are the Pareto front obtained applying Build Allocation algorithm. The red ones are the solutions obtained with random allocation.

As results show Build Allocation algorithm outperforms the random allocation. Red Points are not on the Pareto frontier and they are dominated by blue points. Results indicate that the proposed optimization is better than random allocation. This is symptomatic that Build Allocation algorithm can correctly balance the workload of each simulator/HiL taking into account also tests priorities.

Finally, we tested our solution using two fitness functions and one constrain. In this case, we want to take into account test dependencies. Two fitness functions aim at minimizing the workload of every simulator/HiL and a constrain aims at locating dependent tests on the same machine.

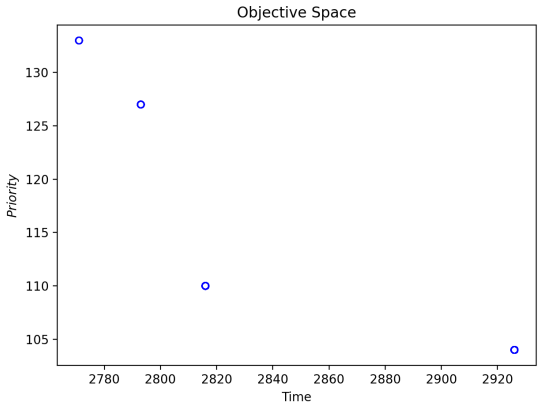


Figure 14: The Pareto-optimal solutions obtained applying the Build Allocation optimization procedure with dependencies constrain.

Figure 14 shows the results after running of Build Allocation algorithm with two optimization functions and one constrain taking into account test dependencies. It shows the Pareto front of all efficient solutions. It is composed of 4 best solutions. Figure 14 shows a set of 4 non-dominated solutions. During this test, we involve 3 simulators/HiL.

Figure 15 shows the results achieved by two different solutions taking into account dependencies constrain. Solutions in blue are the Pareto front obtained applying Build Allocation algorithm. The red ones are the solutions obtained with random allocation.

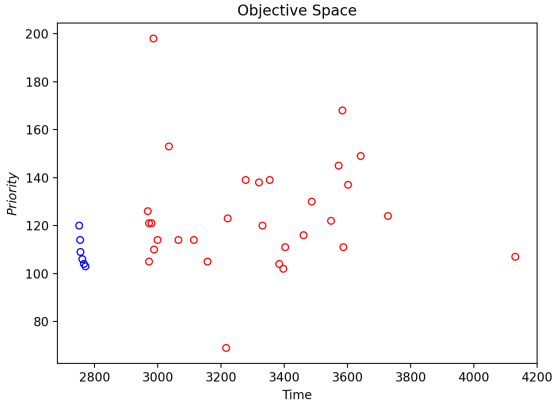


Figure 15: Build Allocation vs Random Allocation comparison results with constrains.

During this run, Build Allocation algorithm outperforms the random allocation. Also, in this case, red points are not on the Pareto frontier and they are dominated by blue points. Results indicate that the proposed optimization is better than random allocation. This is symptomatic that Build Allocation algorithm can correctly balance the workload of each simulator/HiL taking into account: execution times optimization tests priorities and test dependencies.

4.4.2 Synthetic Data Results

This section contains the results achieved using Synthetic Data.

Figure 16 show how synthetic data are distributed. More precisely, it contains test execution times randomly generated starting from the distribution of 72 tests cases belonging to Intelligentia case study.

We generated three distributions with different sizes. Figure 16a shows the execution times of 500 tests cases. Figure 16b shows the execution times of 1000 tests cases. Finally Figure 16c shows the execution times of 2000 tests cases. We use these distributions to further evaluate our approach.

It should be underlined that synthetic distribution sizes are bigger than the one of the real case study. We increase the number of test cases to evaluate our approach also with a substantial amount of tests.

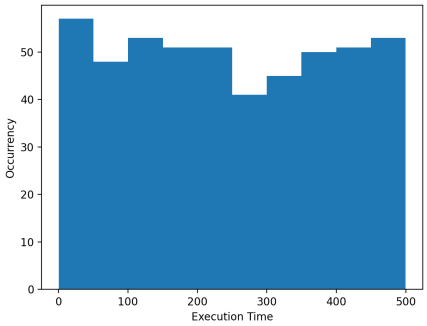
As in the real case study, we compare Build Allocation algorithm with Random allocation. We use two fitness functions aiming at minimizing the workload of every simulator/HiL. We want to take into account test dependencies, thus there is also one constrain aiming at locating dependent tests on the same machine.

For the first test, we use 5 simulators/HiL. We execute for 30 times both solutions: Build Allocation and random allocation.

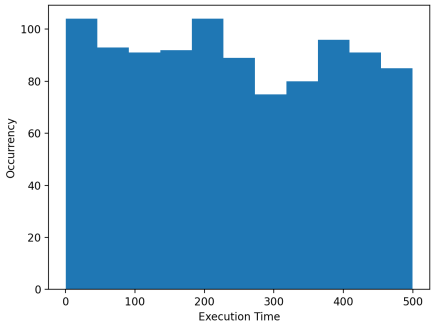
Figure 17 shows the results after running both Build Allocation algorithm and random allocation. It shows the comparison between the two solutions.

Solutions in blue are the Pareto front obtained applying Build Allocation algorithm. The red ones are the solutions obtained with random allocation.

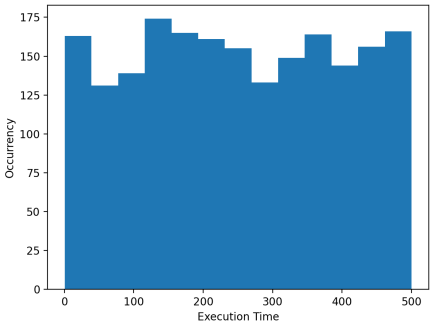
Figure 17a shows the results achieved by two different solutions taking into account the dependencies constrain of 500 tests cases. Figure 17b shows the results achieved by two different solutions using 1000 tests cases. Finally, Figure 17c shows the results achieved by two different solutions over 2000 tests cases.



(a) Tests Execution Time randomly generated (500 test cases).

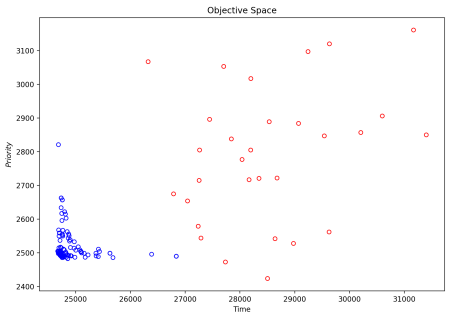


(b) Tests Execution Time randomly generated (1000 test cases).

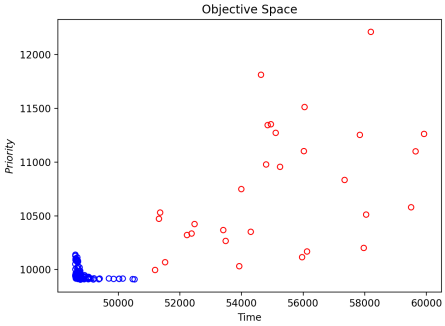


(c) Tests Execution Time randomly generated (2000 test cases).

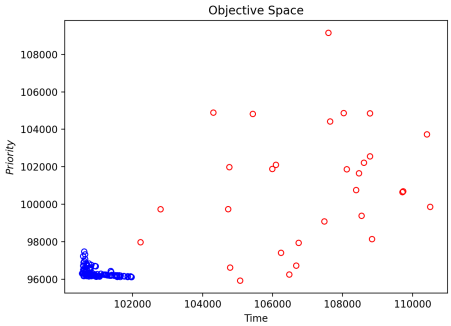
Figure 16: Example Scenario.



(a) 500 test cases.



(b) 1000 test cases.



(c) 2000 test cases.

Figure 17: Build Allocation vs Random Allocation comparison results with constrains between 5 simulators/HiL.

As results show Build Allocation algorithm outperforms the random allocation. Red Points are not on the Pareto frontier and they are dominated by blue points. Results indicate that the proposed optimization is

better than random allocation. This is symptomatic that Build Allocation algorithm can correctly balance the workload of each simulator/HiL taking into account also tests priorities.

Finally, we can also observe that our algorithm is robust when the number of test cases increases.

5 Conclusion

One of the challenges practitioners face, when setting a CI/CD pipeline for CPS development, is the high cost of software and hardware resources (e.g., simulators and real hardware devices).

Furthermore, likely conventional software development, DevOps teams struggle with long build execution time hindering fast feedback rules related to the adoption of a pipeline.

This document proposes an approach aimed at overcoming the challenges described above by optimizing the allocation of jobs/tasks among a limited number of simulators and HiL. Specifically, by using multi-objective genetic algorithms, our approach, named Build Allocation, provides the optimal distribution of tasks/jobs. It carries out two solutions: the first one assigns each job to only one simulator/HiL and the second one duplicates jobs over different simulators/HiL.

The approach has been evaluated using a real case study composed of 72 test cases belonging to a satellite onboard software system that is going to be used on an ESA space mission. Furthermore, we used also synthetic data generated starting from the real one for evaluation purposes. Finally, we compared the obtained results with the ones achieved using a random allocation approach.

Preliminary results are promising: Build Allocation algorithm outperforms random allocation of jobs/tasks between different simulators/HiL.

In future work, we plan to compare our approach with greedy algorithms, e.g. integer programming. Greedy algorithms make a locally-optimal choice leading to a globally-optimal solution. We aim to use multi-objective optimization and we plan to compare the mono-objective solution with greedy ones since it is always possible to substitute a genetic algorithm optimizing only one function with a greedy algorithm.

References

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. A machine learning approach to improve the detection of ci skip commits. *IEEE Transactions on Software Engineering*, 2020.

[2] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 143–154. IEEE, 2018.

[3] Iyad Alazzam and Khalid M O Nahar. Combined source code approach for test case prioritization. In *Proceedings of the 2018 International Conference on Information Science and System*, pages 12–15, 2018.

[4] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. Buildfast: history-aware build outcome prediction for fast feedback and reduced cost in continuous integration. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–53. IEEE, 2020.

[5] John Clarke, Jose Javier Dolado, Mark Harman, Rob Hierons, Bryan Jones, Mary Lumkin, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, et al. Reformulating software engineering as a search problem. *IEE Proceedings-software*, 150(3):161–175, 2003.

[6] Kalyanmoy Deb and Debayan Deb. Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing*, 4(1):1–28, 2014.

[7] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[8] Massimiliano Di Penta, Mark Harman, and Giuliano Antoniol. The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study. *Software: Practice and Experience*, 41(5):495–519, 2011.

[9] Paul M. Duvall. Continuous integration. patterns and antipatterns. *DZone refcard #84*, 2010.

[10] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–245, 2014.

[11] Erik Flores-García, Goo-Young Kim, Jinho Yang, Magnus Wiktorsson, and Sang Do Noh. Analyzing the characteristics of digital twin and discrete event simulation in cyber physical systems. In *Advances in Production Management Systems. Towards Smart and Digital Manufacturing*, volume 592 of *IFIP Advances in Information and Communication Technology*, pages 238–244. Springer, 2020.

[12] David E Goldberg and John Henry Holland. Genetic algorithms and machine learning. 1988.

[13] Foyzul Hassan and Xiaoyin Wang. Change-aware build prediction model for stall avoidance in continuous integration. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 157–162. IEEE, 2017.

[14] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[15] Yusen Li, Xueyan Tang, and Wentong Cai. Dynamic bin packing for on-demand cloud resource allocation. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):157–170, 2015.

[16] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*, pages 688–698, 2018.

[17] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. Test case prioritization for continuous regression testing: An industrial case study. In *2013 IEEE International Conference on Software Maintenance*, pages 540–543. IEEE, 2013.

[18] Cong Pan and Michael Pradel. Continuous test suite failure prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 553–565, 2021.

[19] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. Predicting continuous integration build failures using evolutionary search. *Information and Software Technology*, 128:106392, 2020.

[20] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 12–22, 2017.

[21] Zhaolin Wu, Yang Yang, Zheng Li, and Ruilian Zhao. A time window based reinforcement learning reward for test case prioritization in continuous integration. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, pages 1–6, 2019.

[22] Rajaa Vikhram Yohanandhan, Rajvikram Madurai Elavarasan, Premkumar Manoharan, and Lucian Mihet-Popa. Cyber-physical power system (CPPS): A review on modeling, simulation, and analysis with cyber security applications. *IEEE Access*, 8:151019–151064, 2020.

[23] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25(2):1095–1135, 2020.