Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Lightweight Dynamic Build Batching Algorithms for Continuous Integration

Divya M. Kamath · Bram Adams · Ahmed E. Hassan

Received: date / Accepted: date

Abstract Continuous Integration (CI) is a widely adopted process in software engineering that virtually merges developers' pull requests with the code base to perform builds and tests essential for quality assurance. CI, however, is also an expensive process, due to the large number of pull requests that are pushed by developers on a daily basis. To reduce the cost of CI, many companies adopt batching algorithms that combine several commits into a single build, either with a fixed or a dynamically evolving batch size. While a successful, dynamic batching technique has been proposed by earlier work, we propose a fully online, more flexible dynamic batching technique that can be configured on-the-fly and updates batch sizes only based on the outcome of the previous batch build. Empirical evaluation on 286,848 commits from 50 open-source projects using TravisCI shows that our lightweight batching technique can perform equally well to the more complex state-of-the-art batching techniques, and save a median of 4.75% more builds than static batching techniques.

Keywords Software build · Build avoidance heuristic · Software analysis · Empirical study

1 Introduction

Continuous Integration (CI) [8] is a popular quality assurance process used by countless software organizations according to which a pull request made by a developer is integrated virtually with the current snapshot of a code base in order

D.M. Kamath
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: divya.kamath@queensu.ca

B. Adams
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: bram.adams@queensu.ca

A.E. Hassan
School of Computing, Queen's University, Kingston, ON, Canada
E-mail: hassan@queensu.ca

to build (i.e., compile and test) the changed code. This is typically done before or during the code review process, when any issues identified during CI can still easily be fixed. To support this process, countless CI automation servers (e.g., GitHub Actions, Azure DevOps Server) have been built and are widely available to software organizations.

However, CI is a costly process in terms of time, energy and computing resources. An individual build can take hours to complete [31], thereby affecting developer productivity. Furthermore, for a given code change one often needs to run multiple builds in parallel, for instance to support different platforms or to evaluate the quality of different configurations of a given product. Even large companies like Google and Mozilla report their CI costs in millions of dollars annually [14].

Hence, attempts have been made to reduce the cost of CI builds, both in terms of scheduling less builds [2,5,14] and in terms of making scheduled builds faster [9]. We focus on the former line of research, where researchers have worked on (1) AI-based approaches that decide whether to schedule a build by predicting whether it will fail, and on (2) less complex rule-based approaches. While the former AI-based techniques have shown to be effective, they also bring additional costs and disadvantages to the table, such as the cost of gathering data and training a model, coupled with the cost of retraining the model due to data drift and the potential cost of prediction errors.

Hence, in this study, we focus particularly on rule-based heuristics, i.e., CI-Skip and Batching. CI-Skip is a rule-based technique that defines the characteristics of commits that can be CI-Skipped, i.e., can be integrated without scheduling any build. This technique focuses on eliminating builds for simple commits that would trigger the CI process unnecessarily. Those include commits that touch documentation, source code and meta files, commits that only modify source code comments or format the source code by adding newlines and/or spaces, and commits that are related to version preparation.

Batching is the processing of building multiple incoming commits together in a single build. Although it was first studied empirically by Najafi et al. [24], they indicate that the technique was in use long before by a variety of companies and open-source organizations. Initial batching techniques used a fixed number of commits as batch size. If the batch build was unsuccessful, diverse fallback algorithms like BatchBisect, Batch4, BatchDivide4 and BatchStop4 were used to identify the individual commits responsible for the batch failure.

While effective and lightweight, static batching is not recommended to use in projects that have many failing commits, since it is likely that many failing commits could be grouped into a single batch (or worse, all failing commits in the batch) in such projects, leading to a high build overhead in detecting each failing commit. Instead, if failing commits are anticipated, reducing the batch size can help to reduce the build overhead. Bavand et al. [4] used this principle to introduce the dynamic batching technique, which adapts to different batch sizes to be used in the same project. This technique uses a mathematical model that uses weighted smoothing and failure rate to suggest the most optimal batch size to be used for the next batch. If a batch build failed, the same fallback algorithms as for static batching are then used to identify each failing commit.

However, the dynamic batching algorithm is based on a fixed lookup table generated offline. It stores the value of ideal (project-independent) batch sizes to be used for a range of weighted failure rate values. To modify the current batch behaviour, or customize it to a new project, engineers would have to compute or simulate specialized statistical values replacing the current lookup table, which is not a trivial endeavour.

In this work, we present a technique based upon dynamic batching that can provide at least similar build savings to Bavand et al.'s technique [4], while also providing build engineers with the capability to modify batching practices on-the-fly, only based on the previous batch's failure rate. Our Lightweight dynamic batching (LWD) algorithm is based on simple, lightweight (including random) techniques run after every build to decide the batch size of the next build. We also perform the first empirical study that combines batching with CI-Skip rules, comparing the performance with the individual batching techniques.

Our large-scale empirical study on 286,848 commits from 50 open-source projects collected from the TravisTorrent data repository measures the percentage of builds saved by applying each of the existing and new heuristics (static batching, state-of-the-art dynamic batching, LWD and CI-Skip) to address the following three research questions:

– *RQ1: What is the best LWD approach to dynamically update the batch size?* – We evaluate all our lightweight batch updating LWD techniques against each other to identify the most effective techniques amongst them.
– *RQ2: How does LWD perform in comparison to state-of-the-art static and dynamic batching techniques?* – We compare the performance of our new heuristic against existing batching techniques to identify the most effective heuristic for build scheduling reduction.
– *RQ3: Does the addition of CI-Skip rules make batching more effective?* – We combine the studied batching techniques with CI-Skip to evaluate the potential reduction in build scheduling when combining the two rule-based heuristics.

We summarize below our major study contributions:

– Our LWD algorithms can save a median of 7.75% more builds than state-of-the-art static batching algorithms.
– The performance of the LWD algorithms is equally high as that of state-of-the-art dynamic batching.
– Although CI-Skip is effective in pruning out harmless commits from CI builds, it does not significantly improve on the performance of dynamic batching algorithms.
– Our study's replication package[1] is available online.

We organized the remainder of this paper as follows. Section 2 discusses background information and related work. Section 3 defines the existing and new heuristics discussed in this study. Section 4 introduces our study methodology, including the study goal, research questions, and study steps. Sections 5, 6, and 7 present our study results per research question. Section 8 discusses threats to the validity of our empirical study. Finally, Section 9 concludes this paper and suggests future work.

[1] https://github.com/divyamadhav/dynamic_batching

2 Background and Related Work

2.1 Related Work

2.1.1 Reducing CI Costs

Large software companies like Google and Facebook have a massive number of commits to compile everyday. As stated by Najafi et al. [24], software testing hence becomes one of the costliest stages of software development. To reduce CI costs, researchers introduced techniques that tackle this problem from different angles, such as reducing the number of builds to be scheduled, speeding up scheduled builds using faster build systems and architectures, and reducing the number of test cases to be run. These techniques include the widely popular Test Case Prioritization (TCP), Regression Test Selection (RTS), build priorization, commit skipping and commit batching.

Various authors have explored approaches for Test Case Prioritization (TCP) and Regression Test Selection (RTS), aiming to reduce test duration in CI [6,9,10, 13,21,27]. Kazmi et al. [20] presented a literature survey of 47 empirical studies of RTS techniques like learning-based regression and mining, their study design and the cost models of studied RTS techniques. Elbaum et al. present cost-effective techniques that use time windows to track executed test suites and apply RTS and/or TCP during CI testing. Jin et al. [15, 16] also evaluated 10 selection and prioritization strategies on build and test level granularities to understand design decisions that could help to save the cost of CI. In parallel, companies like Microsoft, Facebook and Google have also introduced distributed build tools like Buck, CloudBuild [11], Bazel [22] that reduce build duration by performing incremental builds. Approaches involving prioritization, test case reduction or build tool optimization tackle the CI costs once the CI builds are scheduled. On the other hand, approaches like commit batching and commit skipping studied by this paper aim to reduce the number of builds actually scheduled.

At the level of build scheduling, batch testing and bisection are employed when resources for expensive build and test operations are scarce. The basic principle combines together a finite number of individual commits into a single build/test. Only if the build/test fails, the commit(s) responsible for the failure must be identified through additional builds based on a so-called fallback algorithm. Hence, commit batching requires choosing a batch size (i.e., finite number of commits) and a fallback algorithm (to identify culprit commits). Najafi et al. [24] presented a study on identifying patterns of commit grouping and bisection. They identified that batching and bisection are effective when performed in batch sizes of up to 9 builds. Beheshtian et al. [5] presented and examined 4 commit bisection techniques and 2 risk modeling techniques to perform batching in CI. Furthermore, Bavand et al. [4] presented a study on dynamic commit batching, which uses a non-trivial mathematical model to determine and evolve the ideal batch size to be used during CI. In our study, we explore and evaluate the use of simpler, arithmetic and random operations to dynamically determine batch sizes to be used during CI. We compare our study results against the above works to provide build engineers with an in-depth insight into commit batching and bisection.

Commit skipping is a build scheduling technique trying to determine which builds are useful to schedule, and which ones are redundant (i.e., should not be scheduled) during CI testing. After identifying 1,813 commits that were manually labeled by developers to be skipped from CI testing, Abdalkareem et al. [2] presented CI-Skip rules able to identify commits for which a build is redundant, then presented a rule-based technique to automatically detect and label CI-Skip commits. Furthering their study, Abdalkareem et al. [1] presented a machine learning approach to improve the detection of CI-Skip commits. This study identified that the number of developers, content of commit messages along with the CI-Skip rules are important factors for detecting CI-Skip commits. They extracted 23 features from the historical data of 10 software repositories and built a decision tree classifier to detect these commits. Jin et al. [17] further studied the cost-saving ability and the safety of CI-Skip rules with respect to the proportion of failing builds that they can identify and determined that the rules are not completely safe to use, as they skip some failing builds too. They further presented a safer, complementary collection of CI-Run rules that are used in their novel PreciseBuildSkip technique to predict build failures.

Another approach to skip commits involves training machine learning algorithms to predict the outcome of a CI build. If a commit is predicted to build successfully, it is merged into the code base directly, without an actual build. In contrast, if the build of a commit is predicted to fail, the build is scheduled in order to identify the error. Jin et al. [14] introduced SmartBuildSkip, a random forest classifier for build skipping able to save between 30%-61% builds. Later on, they combined their work on SmartBuildSkip and PreciseBuildSkip to aggregate the predictions of both techniques to provide build/skip predictions with more confidence [18]. In our study, we avoid the use of machine learning models in our study, and instead apply the CI-Skip rules along with our studied batching techniques to examine if the two lighterweight rule-based techniques can be combined to achieve better build savings.

2.1.2 Local vs. Global Models in Software Analytics

The Occam's razor philosophy states that, given a choice between two competing hypotheses, the simpler theory should be selected [29]. The principle has been widely employed in previous work in Computer Science, in the contexts of machine learning, project management, big data and service oriented architectures among others. As an obvious example, Fan et al. [12] compared the CLIFF&MORPH models (privacy preserving, data-sharing algorithms) with the simple, unsupervised ManualDown model, which considers a file to be defect-prone if it is amongst the top N largest files that were changed in a given time period. Their empirical study showed that this simple model could perform at least as good, often even better cross-company defect prediction than the more complex CLIFF&MORPH.

Another take on Occam's Razor is provided by the use of local vs. global models for various software engineering tasks, where local models only consider limited knowledge known locally, while global models require a global view of a given system. For instance, Pinciroli et al. [25] presented Buzz, a programming language for robot swarms. The language allowed to define swarm robot behaviours with respect to either a single robot (local) or the overall swarm (global). Using their Buzz domain specific language, the authors show that a variety of global swarm behaviours can be obtained solely by defining a given type of robot's individual interactions with its immediate neighbourhood. Using a global, centralized view of the swarm for programming the swarm's behaviour often is not required, and might even lead to more brittle, complex algorithms.

Similarly, Menzies et al. [23] found, with respect to effort estimation and defect prediction, that localized learning of data across a population can provide important findings that differ in comparison to lessons learnt from global data. In particular, since they can specialize to more homogeneous groups of files, local models tend to be more simple, while also performing better.

Finally, Zaha et al. [30] also discussed the tradeoffs between the use of local and global knowledge in service-oriented architecture (SOA). While a global picture of service interactions is useful for larger service oriented systems, these systems can not properly replicate local behaviour. Hence they define an algorithm to generate more simple, local models of system interactions from a global model to gain local knowledge on system interactions.

Similar to the above work, we also focus on simple, local algorithms to dynamically update the size of a batch during CI, relying solely on (1) the previous batch's failure rate and (2) simple arithmetic.

2.2 Background

This section revisits the static and dynamic algorithms proposed in prior work, which we use as baselines in this paper.

2.2.1 TestAll Baseline

TestAll is a performance measuring baseline used by Bavand et al. [4] and Beheshtian et al. [5] that we follow in this work as well. In TestAll, all incoming commits are compiled individually during CI, without any batching. Since TestAll does not save any build at all, we measure the percentage of builds saved by the studied heuristics with reference to TestAll - i.e., Heuristic A saves X% of builds w.r.t TestAll.

2.2.2 Static Batching

During static batching, a finite number of incoming commits are combined into a single batch for build execution [5], see Algorithm 1. If a batch of 'N' commits compiles successfully, it saves N-1 builds from being executed. However, if the batch build is unsuccessful, fallback algorithms like BatchBisect, BatchStop4, BatchDivide4, etc. are used to identify the culprit commit(s). In pre-merge testing, batching can be applied by grouping together changes submitted in parallel by different developers. As the changes of each commit typically are independent of other commits in the batch, the entire batch would fail if a single commit in the batch fails. On the other hand, during post-merge testing, changes from developer commits have been integrated and tested already, hence are more likely to pass the build. If the batch consists of a (few) failing commit followed by a successful commit, the entire batch is likely to build successfully.

Figure 1 shows the BatchBisect algorithm for a batch size 4. The commits are grouped according to their chronological order, batching together every 4 commits. Since the entire batch build would fail if it consists of at least one failing commit, the batch is bisected into two sub-batches for further examination. As shown, we see a total of 8 builds required for building 8 commits, making a savings of 0 builds. Algorithms 2-4 summarize the steps of these studied fallback algorithms.

In BatchBisect, culprit commits are identified by halving the batch size (bisection) until a batch size of 1 commit is reached for failing sub-batches. However, with BatchStop4 and BatchDivide4, bisection is performed only until the batch size of 4 is reached, after which all 4 commits of a failing sub-batch are built individually to reveal any failing commits. BatchDivide4 further generalizes BatchStop4, by first grouping one batch into 4 groups of similar sizes, then implementing BatchStop4 on each batch to reveal any culprit commit(s).

Algorithm 1 Batch* Driver

```
1: procedure FILLBATCH(batchSize, incomingCommit)
2:     $i = 0$
3:     $batch = []$
4:     $fails = []$
5:     while $i < batchSize$ do
6:         $batch.append(incomingCommit)$
7:         $i \leftarrow i + 1$
8:     end while
9:     $outcome \leftarrow BatchBisect(batch, fails)$        ▷ or BatchStop4 or BatchDivide4
10: end procedure
```

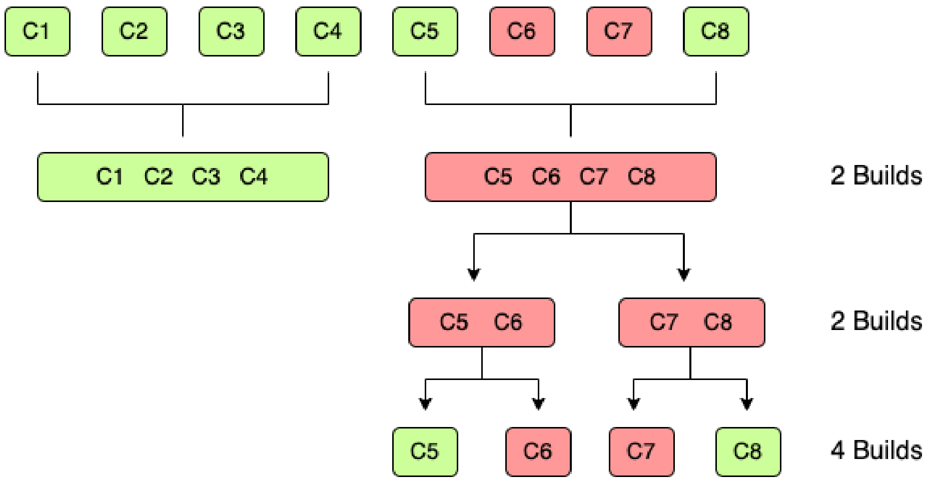


Fig. 1: This figure illustrates the number of builds required by the BatchBisect commit grouping algorithm (batch size 4). In the figure, red boxes signify failing builds, while green boxes signify successful builds. Across the 8 commits, BatchBisect will require a total of 8 builds (2+2+4) to be scheduled due to the bisection used for the failed second batch build.

Algorithm 2 BatchStop4 (Beheshtian et al. [5])

```
1: procedure BATCHSTOP4($batch$,$fails$)
2:     if $len(batch) <= 4$ then
3:         $i \leftarrow 0$
4:         while $i < batch\_size$ do
5:             $build\_outcome = Build(batch[i])$
6:             if $build\_outcome == False$ then
7:                 $fails.extend(batch[i])$
8:             end if
9:         end while
10:        $return$
11:    end if
12:    $outcome \leftarrow Build(batch)$
13:    if $outcome == False$ then
14:        $half = len(batch)/2$
15:        $BatchStop4(batch[: half], fails)$
16:        $BatchStop4(batch[half :], fails)$
17:    end if
18: end procedure
```

Algorithm 3 BatchBisect (Beheshtian et al. [5])

```
1: procedure BATCHBISECT($batch, fails$)
2:     if $len(batch) == 1$ then
3:         $build\_outcome = Build(batch)$
4:         if $build\_outcome == False$ then
5:             fails.extend(batch)
6:         end if
7:         $return$
8:     end if
9:     $outcome \leftarrow Build(batch)$
10:    if $outcome == False$ then
11:        $half = len(batch)/2$
12:        $BatchBisect(batch[: half], fails)$
13:        $BatchBisect(batch[half :], fails)$
14:    end if
15: end procedure
```

Algorithm 4 BatchDivide4 (Bavand et al. [4])

```
1: procedure BATCHDIVIDE4($batch$,$fails$)
2:     if $len(batch) <= 10$ then
3:         return BatchStop4(batch)
4:     else
5:         $limit \leftarrow len(batch) \div 4$
6:         $batch\_1 \leftarrow batch[: limit]$
7:         $outcome \leftarrow BatchStop4(batch\_1, fails)$
8:         $batch\_2 \leftarrow batch[limit : 2 * limit]$
9:         $outcome \leftarrow outcome$ & $BatchStop4(batch\_2, fails)$
10:        $batch\_3 \leftarrow batch[2 * limit : 3 * limit]$
11:        $outcome \leftarrow outcome$ & $BatchStop4(batch\_3, fails)$
12:        $batch\_4 \leftarrow batch[3 * limit :]$
13:        $outcome \leftarrow outcome$ & $BatchStop4(batch\_4, fails)$
14:    end if
15: end procedure
```

The largest drawback with static batching is the bisection overhead when a compiled batch consists of many failing commits. Since build failures are expected to occur consecutively in CI, intuitively, a stream of failing commits are better compiled individually to avoid the overhead of bisection builds. For instance, let's assume a worst case scenario involving a batch of 8 failing commits. To identify that all commits in the batch are failing, BatchBisect would require to make 15 build executions (see Figure 1). BatchDivide4 and BatchStop4 would require 11 builds each. This is an overhead of $15 - 8 = 7$ and $11 - 8 = 3$ builds, respectively.

2.2.3 Dynamic Batching

Dynamic Batching, introduced by Bavand et al. [4] uses a weighted moving average of build failures to dynamically estimate an optimal batch size that could be used to perform commit batching during CI. The $WeightedFailureRate$ of the last 'C' commits is computed using:

$$WeightedFailureRate = \frac{\sum_{c=1}^{C} Smoothing(c) * IsFailure(c)}{\sum_{c=1}^{C} Smoothing(c)} \quad [4]$$

Here, 'c' is the position of a commit relative to the current commit, while $Smoothing$ is a function that smoothes the weights for each commit. Examples of such a function are 1, $log(c)$, e^c or $\frac{1}{c}$, most of which ensure that newer commits have a higher weight than older commits. $IsFailure$ is a predicate indicating whether commit 'c' is a failing build.

Algorithm 5 describes the steps of the state-of-the-art dynamic batching technique. The technique uses an offline simulation of 100k randomly generated pass/fail occurrences for a range of failure rates between (0, 1) and for varying batch sizes in order to identify the expected number of build executions for each measured combination of build failure rate and batch size. Using this data, the *LookupBatchSize* function is able to return the batch size that expects the minimum number of executions (according to the simulation), for a provided build failure rate. Hence, once the CI and batching process begin, the function *WeightedFailureRate* first computes the build failure rate for a finished batch, using the formula provided above, after which the optimal batch size can then be determined using *LookupBatchSize*.

Once the dynamic batching algorithm forms batches and schedules a build, it relies on the basic fallback algorithms BatchBisect, BatchStop4 or BatchDivide4 to identify culprit commits (if any). Hence, there exist three variants of this technique, each using one of the three fallback algorithms.

This state-of-the-art dynamic technique, while an excellent heuristic saving a median of 43.26% builds, relies on the build outcomes of the last N (i.e., 100, 200, ..., commits) historical commits and allows little flexibility for build engineers to tweak the parameters of the algorithm on-the-fly. In fact, the Monte Carlo simulation determining the ideal weights for historical build failure rates does not depend on a project's specific context. As such, we aim to provide more flexibility and present lightweight techniques that can provide similar build saving outcomes while also using only the most recent build data (i.e., a more local approach).

2.2.4 CI-Skip Rules

The concept of CI-Skip rules is based on the premise that not every commit needs to trigger the CI process [2]. Examples of such commits include commits that only

Algorithm 5 State-of-the-art Dynamic Batching, (Bavand et al. [4])

```
 1: procedure FILLBATCH
 2:     $commit\_outcomes = []$
 3:     while incomingCommit do
 4:         while $i < batchSize$ do
 5:             $batch.append(incomingCommit)$
 6:             $i \leftarrow i + 1$
 7:         end while
 8:         $BatchBisect(batch, fails)$                ▷ or BatchStop4 or BatchDivide4
 9:         $commit\_outcomes.extend(fails)$
10:         $failure\_rate \leftarrow WeightedFailureRate(Smoothing, commit\_outcomes)$
11:         $batchSize \leftarrow LookupBatchSize(failure\_rate)$
12:         $i = 0$                                    ▷ Resetting for next batch
13:         $batch = []$
14:         $fails = []$
15:     end while
16: end procedure
```

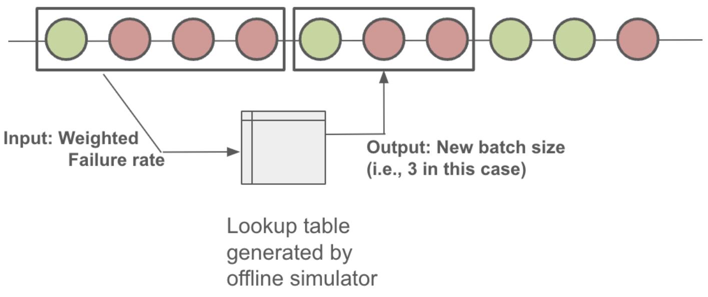

Input: Weighted Failure rate
Output: New batch size (i.e., 3 in this case)
Lookup table generated by offline simulator

Fig. 2: This figure illustrates the Baseline Dynamic Batching algorithm. After each batch build, the algorithm consults the lookup table produced by the offline simulator, which uses the weighted failure rate to output the batch size most suitable for the next batch build.

modify source code comments or commits that only modify the documentation of a project. Abdalkareem et al. introduced CI-Skip rules after examining 1,813 Java commits that developers explicitly marked to be skipped from the CI processes. These commits are hence named CI-Skip commits. The rule-based technique has 5 rules to detect CI-Skip commits [2]:

- Changes that touch only documentation and non-source code files
- Changes related to preparing releases (version preparation) such as commits modifying version numbers in build configuration files
- Changes that only modify source code comments
- Changes that touch meta files such as .ignore files or image/media files like .png, .mp3
- Changes that format source code

Abdalkareem et al. [1] later presented a machine learning approach to detect CI-Skip commits, which remains outside the scope of this study as we are dedicated to the evaluation of rule-based techniques.

3 Lightweight Dynamic Batching Algorithm and Variants (LWD)

Lightweight Dynamic Batching (LWD) is a simple rule-based batching heuristic that can dynamically update batch sizes of traditional batching techniques during CI operations based on local knowledge. It only uses information about the failure rate, i.e., number of failing builds in the most recent batch, to decide the batch size for building upcoming commits. If the majority of commits in the previous batch are failing, then a smaller batch size is used for the current batch, given the high risk of future build failures [26]. Conversely, if the majority of commits in the previous batch are successful, then a larger batch size is used, as the risk of future build failures has been reduced.

The main challenge of an LWD technique is to try to increase or decrease batch sizes at the right moment, such that the batch size can be as large as possible (i.e., minimal number of builds to be run) in between build failures, while as small as possible during a build failure streak (i.e., minimizing the bisection overhead of traditional batching). As such, a variety of LWD variants can be designed differing mainly in the way in which the new batch size is estimated.

This paper presents 5 main LWD variants covering a wide range of techniques for incrementing/decrementing batch sizes. All of these share a few basic principles:

- *Fallback Rule: If the failure rate of the most recent batch is greater than a given X%, the next batch size defaults to 1.* – This rule helps to account for a continuous stream of failing incoming commits. In order to avoid unnecessary bisection builds to identify the failing commits in a batch, we build one commit at a time, influenced by SmartBuildSkip [14], until a successful build is encountered. Once a successful build is encountered, the batch size is increased again based on the LWD variant's batch size updating rules.
- *Retention Rule: Retain batch size if the most recent batch has a successful final commit and failure rate less than Y%.* – This rule helps to account for scattered failing incoming commits. If there were very few failing commits in a batch, eventually succeeded by successful commits, we do not anticipate more failing commits in the next batch.
- *Factor Rule: Every batch size update occurs with respect to a factor.* – Fundamentally, a new batch size is obtained by adding, subtracting, multiplying, or dividing the most recent batch size with a factor to obtain a new batch size. With the exception of Random batch updating, where the next batch size is chosen randomly, all our batch updating techniques have non-zero factors.
- *Customizability Rule: Every batch size updating technique can be customized according to the needs of a project.* – In every LWD technique, there exist customizable elements, such as maximum and minimum batch size, factor, retention limit and fallback limit to be used, to allow the build engineers flexibility to control batching in CI according to their current requirements.

These design decisions add to the LWD approaches' explainability, making it easier for build engineers to understand why the given batch size is the most optimal one to use for the next batch.

3.1 Lightweight Batch Size Updating Variants

This subsection details our five LWD techniques incorporating the 4 principles presented previously, yielding Linear, Exponential, Random, Mixed and Most Frequently Used (MFU) variants. Of these five, the Linear, Exponential and Random techniques are atomic, i.e., they comprise only one condition to decide the next batch size. The Mixed and MFU variants are composite techniques that go through a series of if-else conditions to decide the next batch size.

Similar to the state-of-the-art dynamic batching technique, the LWD technique also relies on the three fallback algorithms BatchBisect, BatchStop4 and BatchDivide4 to identify culprit commits in a failing batch. Once a batch is formed and build is executed, any culprits of the batch are identified using one of the 3 fallback algorithms. Once culprits are identified (if any), we update the batch size using our five LWD batch updating variants.

Algorithm 6 calls the Failure Case and Success Case procedures depending on the outcome of the previous batch build. If the previous batch consists of all successful commits and the batch were compiled with a single build, the Success Case procedure is called. Otherwise, the algorithm calls the Failure Case procedure.

Algorithm 6 Batching Function

```
1: procedure FillBatch: (incomingCommitFlag)
2:     $i \leftarrow 0$
3:     $batch \leftarrow []$
4:     $fails \leftarrow []$
5:     $batchSize \leftarrow 16$                                ▷ Default starting batch size
6:     if incomingCommitFlag == True then
7:         $incomingCommit \leftarrow$ new commit received
8:         while $i < batchSize$ do
9:             $batch.append(incomingCommit)$
10:            $i \leftarrow i + 1$
11:        end while
12:        $BatchBisect(batch, fails)$                  ▷ or BatchStop4 or BatchDivide4
13:        $FR \leftarrow Length(fails) \div batchSize$          ▷ Measuring Failure Rate
14:        if $FR > 0$ then
15:            $batchSize \leftarrow Failure\_Case(FR)$
16:        else
17:            $batchSize \leftarrow Success\_Case()$
18:        end if
19:    end if
20: end procedure
```

In this study, we identify 39 sub-variants of the LWD batching technique. I.e., the five batch updating techniques form 13 sub-variants, each of which can be used in combination with one of three fallback algorithms. These sub-variants are listed in Table 1.

3.1.1 Atomic Batch Size Updating: Linear, Exponential, Random

Algorithms 7, 8, 9 details the steps of the atomic batch size updating techniques. In these algorithms, the elements fallback_limit (implements the Fallback Rule),

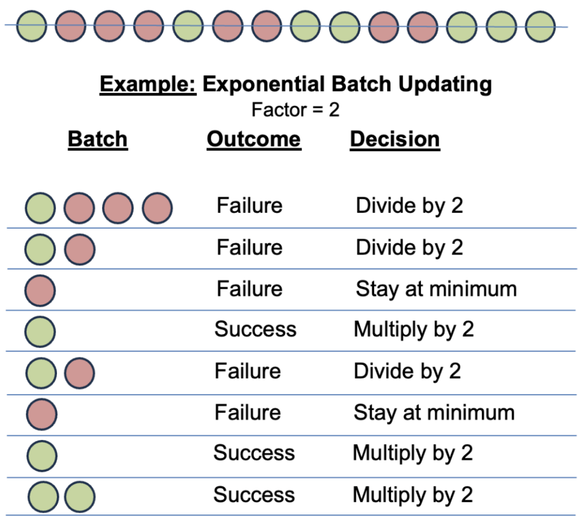
Table 1: Batch size update rules for 3 atomic and 2 composite LWD variants

Variant		Factors	New Batch Size	
Base	Sub-Variant		Success Case	Failure Case
Linear		1,2,3,4	current+factor	current-factor
Exponential		2,3	current*factor	current ÷ factor
Random	Linear		current + F F = random(current, maximum)	current - F F = random(minimum, current)
	Exponential		current * F F = random(2, 4)	current ÷ F F = random(minimum, current)
	Jump		random(current, maximum)	random(minimum, current)
Mixed		2,3	current*factor	current-factor or current÷factor or minimum
MFU		2,3	current*factor	current-factor or current÷factor or MFU batchSize

retention_limit (implements the Retention Rule), factor (implements the Factor Rule), minimum and maximum can be customized according to the needs of the project. As depicted in lines 2, 4 and 6, these techniques use the failure rate of the previous batch to decide whether to retain the previous batch size, fall back to minimum batch size or update the batch size. To update the batch size in lines 9 and 17, they add, subtract, multiply or divide the current batch size by 'factor' (which is either constant or randomly chosen) as shown in Algorithms 7, 8 and 9.

The atomic batch updating techniques are designed to allow either aggressive or more lenient updates to batch sizes. For example, if the current batch size used is 4 builds, the linear variant with factors of $1, 2, 3, 4$ can increment/decrement a batch by the specified amount, making smaller and more lenient changes in batch size. On the other hand, in projects with a high velocity of incoming commits, the batch size could more quickly expand the batch size from 16 to 48 by using an Exponential LWD with a factor of 3. If build engineers do not have a concrete idea of the factor values needed or LWD technique to use, they could use the random batch updation technique as it uses random factors and batch sizes. Additionally, by choosing different sub-variants of random batching such as Jump, Linear or Exponential, one can obtain more control on the nature of build savings.

We choose reasonable values for 'factor' to avoid making too large batch sizes (e.g., batch sizes exceeding 50 builds). To do so, we limit the factors of Exponential variant to $\{2, 3\}$. Similarly, factors of the Random-Exponential sub-variant are limited to $\{2, 3, 4\}$. We maintained a 20% retention limit and 40% fallback limit in our experiments. In other words, if less than 20% of the commits in the previous batch have failed, then the batch size is reused for the next batch, and if more than 40% of commits of the previous batch failed, we reset the batch size to minimum (i.e., 1). We also set the maximum allowed size of the batch to be 16 commits, so



Example: Exponential Batch Updating
Factor = 2

Batch	Outcome	Decision
🟢🔴🔴🔴	Failure	Divide by 2
🟢🔴	Failure	Divide by 2
🔴	Failure	Stay at minimum
🟢	Success	Multiply by 2
🟢🔴	Failure	Divide by 2
🔴	Failure	Stay at minimum
🟢	Success	Multiply by 2
🟢🟢	Success	Multiply by 2

Fig. 3: This figure illustrates the Exponential Lightweight Dynamic Batching variant. After each batch build, depending on the failure rate of the most recent batch, the algorithm increases or decreases the batch size for the next build by a factor of 2.

as to avoid making larger batch sizes than 50 builds while using the Exponential technique. We used a low retention limit to avoid the risk of encountering more build fails in the subsequent batches. A moderate fallback limit allowed to ensure adequate build savings, as a lower value would frequently reset the batch size to 1, hindering build saving efforts and a higher value could lead us to include more failing commits in the next batch.

3.1.2 Composite Batch Size Updating: Mixed, MFU

The composite batch size updating techniques use an if-else-if branch to decide the next batch size to use in CI. These approaches help to form more conservative batch updating techniques that can decide how much a batch size needs to be reduced. The difference between our two composite LWD variants boils down to their ability to handle large failure rates. While the Mixed LWD variant immediately falls back to the minimum batch size, the MFU algorithm will default to the batch size that was most frequently used thus far for this project. Composite batch updating variants were introduced to provide more balance between Linear and Exponential atomic techniques where, depending on the acceptable level of build failures in the project, engineers can configure to either drastically or conservatively update batch sizes.

Algorithms 10 and 11 present the Mixed and MFU variants. In these algorithms, elements *fallback_limit, min_limit, med_limit, max_limit, factor, minimum* and *maximum* are customizable according to the needs of the project. Each of these limits are tested individually as well as along with the criterion that the previous batch ended with a successful commit (to ensure that we are not in the middle of a

Algorithm 7 Linear Batch updating algorithm

```
1: procedure FAILURE CASE : (FR = failure_rate)
2:     if ($FR > fallback\_limit$) then                                    ▷ Fallback Rule
3:         return minimum
4:     else if ($FR < retention\_limit$) && ($last\_commit = Success$) then  ▷ Retention Rule
5:         $return(current\_size)$
6:     else if $current\_size < factor$ then
7:         return minimum
8:     else
9:         $new\_size \leftarrow current\_size - factor$                      ▷ Factor Rule
10:        return new_size
11:    end if
12: end procedure

13: procedure SUCCESS CASE
14:    if $current\_size > maximum$ then                                   ▷ Batch size should not
15:        $return(current\_size)$                                          ▷ be beyond $max + factor$
16:    else
17:        $new\_size \leftarrow current\_size + factor$
18:        return new_size
19:    end if
20: end procedure
```

Algorithm 8 Exponential Batch updating algorithm

```
1: procedure FAILURE CASE : (FR = failure_rate)
2:     if ($FR > fallback\_limit$) then                                    ▷ Fallback Rule
3:         return minimum
4:     else if ($FR < retention\_limit$) && ($last\_commit = Success$) then  ▷ Retention Rule
5:         $return(current\_size)$
6:     else if $current\_size < factor$ then
7:         return minimum
8:     else
9:         $new\_size \leftarrow current\_size \div factor$                   ▷ Factor Rule
10:        return new_size
11:    end if
12: end procedure

13: procedure SUCCESS CASE
14:    if $current\_size > maximum$ then                                   ▷ Batch size should not
15:        $return(current\_size)$                                          ▷ be beyond $max \times factor$
16:    else
17:        $new\_size \leftarrow current\_size \times factor$
18:        return new_size
19:    end if
20: end procedure
```

series of failing commits). If the *min_limit* is not yet reached, we retain the current batch size in the case of the Mixed batch updating technique. Consequently, for Mixed and MFU techniques we update the batch size cautiously by subtracting $factor$ if the *med_limit* is not reached, but more aggressively divide by $factor$ if the *med_limit* is exceeded. In our experiments, we used the values of 10, 20 and 50 for the *min_limit*, *med_limit* and *max_limit* respectively. To avoid making large batch sizes, we limit the $factor$ used in the Composite batch updating variants to

Algorithm 9 Random Batch updating algorithm

```
 1: procedure FAILURE CASE : (FR = failure_rate)
 2:     if ($FR > fallback\_limit$) then                                   ▷ Fallback Rule
 3:         return minimum
 4:     else if ($FR < retention\_limit$) && ($last\_commit = Success$) then  ▷ Retention Rule
 5:         return($current\_size$)
 6:     else if $current\_size < factor$ then
 7:         return minimum
 8:     else
 9:         $new\_size \leftarrow current\_size - / \div random$           ▷ Factor Rule according to Table 1
10:         return $new\_size$
11:     end if
12: end procedure

13: procedure SUCCESS CASE
14:     if $current\_size > maximum$ then                               ▷ Batch size should not
15:         return($current\_size$)                                    ▷ be beyond $max + / \times factor$
16:     else
17:         $new\_size \leftarrow current\_size + / \times random$         ▷ Factor Rule according to Table 1
18:         return $new\_size$
19:     end if
20: end procedure
```

{2, 3} only. We also set minimum batch size to be 1 commit and maximum batch size to be 16 commits. Similar to Atomic batch updating variants in Section 3.1.2, we set low retention limits and moderate fallback limits to attempt adequate build savings.

3.2 Use Case of Lightweight Dynamic Batching

The lightweight dynamic batching technique is designed to enable build engineers to customize the configuration of their CI batching. Build engineers can choose any of the 5 methods to save builds during CI. For each of these techniques, the engineers can set the values of the fallback limit, retention limit according to the cost/benefit tolerance of the project. The absence of offline elements in LWD also allows engineers to update parameters that modify build behaviour on-the-fly, only leveraging local knowledge of the previous batch's outcome.

The choice of LWD variant and configuration can be adapted to a project's characteristics and development cycle. For instance, if a project needs to identify failing commits early and cost is not an issue, then the fallback limit and retention limits can be set low. Similarly, if the project needs to save the cost of executing builds, then the fallback and retention limit can be set high. Similarly, developers can also choose to set higher factor values (i.e., greater than 3) to allow larger batch sizes to be used in build execution.

Additionally, during the development cycle, when new features are being added into software, feedback is essential to understand what code changes are ready and which ones need more work. Engineers can choose to set more conservative limits (i.e., lower batch sizes) to be able to learn build outcomes as early as possible. However, in between development cycles, where incoming commits are mostly

Algorithm 10 Mixed Batch Size Updating Techniques

```
 1: procedure FAILURE CASE : (FR = failure_rate)
 2:     if ($FR > fallback\_limit$) then                                  ▷ Fallback Rule
 3:         return minimum
 4:     end if
 5:     if $current\_size < factor$ then
 6:         return (minimum)
 7:     else if ($FR < retention\_limit$) && ($lastcommit = Success$) then  ▷ Retention Rule
 8:         return (current_size)
 9:     else if ($FR < med\_limit$) && ($lastcommit = Success$) then
10:         return (current_size - factor)
11:     else if ($FR < max\_limit$) && ($lastcommit = Success$) then
12:         return (current_size ÷ factor)
13:     else
14:         return minimum
15:     end if
16: end procedure


17: procedure SUCCESS CASE
18:     if $current\_size > maximum$ then
19:         return (maximum)
20:     else
21:         return min(current_size*factor, 16)
22:     end if
23: end procedure
```

Algorithm 11 MFU Batch Size Updating Techniques

```
 1: procedure FAILURE CASE : (FR = failure_rate)
 2:     if ($FR > fallback\_limit$) then                                  ▷ Fallback Rule
 3:         return minimum
 4:     end if
 5:     if $current\_size < factor$ then
 6:         return (minimum)
 7:     else if ($FR < retention\_limit$) && ($lastcommit = Success$) then  ▷ Retention Rule
 8:         return (current_size)
 9:     else if ($FR < med\_limit$) && ($lastcommit = Success$) then
10:         return (current_size - factor)
11:     else if ($FR < max\_limit$) && ($lastcommit = Success$) then
12:         return (current_size ÷ factor)
13:     else
14:         return MFU(batch_size)
15:     end if
16: end procedure


17: procedure SUCCESS CASE
18:     if $current\_size > maximum$ then
19:         return (maximum)
20:     else
21:         return min(current_size*factor, 16)
22:     end if
23: end procedure
```

addressing bug fixes and code maintenance, higher batch sizes can be used to save builds.

From time to time, projects could also monitor the performance of the selected (and possibly customized) LWD technique on the project's historical CI data. If the customizations were not able to improve the approach's performance, they might opt to switch to a different LWD approach altogether instead of further customizing the current approach. They might also decide to make such switches when disruptive changes are expected in the development cycle, depending on the development cycle..

Our technique improves upon the existing dynamic batching algorithm by avoiding the need to pre-compute a look-up table for batch sizes or to frequently recalculate the weighted failure rate. The lookup table of Bavand et al.'s existing dynamic batching technique forces build engineers to use the same settings throughout the entire project lifetime, leaving them unable to make any other project-specific customizations. Similarly, with state of the art static batching, a constant batch size is used for every batch that is built in the project, irrespective of the results of the most recent build. Although effective, different phases of software engineering like development, quality assurance and project maintenance are likely to have different CI needs. This is why the lightweight dynamic batching algorithm does not require a look-up table, but dynamically adapts the batch sizes using intuitive arithmetic formulae.

One of the major advantages of LWD is its customizability, as illustrated above. The customizability rule of LWD helps to adjust CI parameters to fit the context of a project (which might change over time), along with the CI needs according to the software engineering phase. Build engineers can customize limits of batch size retention, fallback limits or decide maximum/minimum batch size allowed during CI, depending on the priorities of their company. LWD's factor rule helps to adjust the frequency of scheduling builds by controlling the rate of increase/decrease of batch sizes. For instance, after a successful batch compilation, factor 3 suggests a bigger batch size than factor 2 for the next build. In a situation where consecutive incoming commits are expected to be successful, larger batch sizes can reduce the number of builds required and subsequently reduce CI cost. At the same time, LWD's fallback rule can help abruptly reset the batch size to 1 when many failing commits are incoming. This reduces the overhead of bisection builds, which are necessary to identify the commits that are responsible for failing the batch build.

4 Study Methodology

4.1 Study Goal and Research Questions

In order to validate the performance of our LWD variants compared to the baseline batching techniques of Section 2.2, we perform an empirical study. Using the Goal/Question/Metric template [3], our study goal is as follows: **analyze** the performance of the lightweight dynamic batching variants during Continuous Integration (CI) testing; **for the purpose of** (i) identifying simpler heuristics for build savings and (ii) identifying ideal batch sizes for batching techniques; **with respect to** the percentage of builds saved in comparison to building all commits; **from the point of view of** software engineering researchers with expertise in mining software repositories and CI testing; and **in the context of** state-of-the-art heuristics and open source projects collected from a Travis CI data set.

In particular, we address the following research questions:

RQ1: *What is the best LWD approach to dynamically update the batch size?* – We use this question to compare the proposed linear, exponential, random, mixed and most frequently used (MFU) batch updating variants to identify the technique that saves most builds in CI.

RQ2: *How does LWD perform in comparison to state-of-the-art static and dynamic batching techniques?* – We aim to identify the technique that is not only simple to use but can also save the most builds in CI.

RQ3: *Does the addition of CI-Skip rules make batching more effective?* – We empirically evaluate the existing rule-based CI-Skip technique both in combination with our LWD batching variants and state-of-the-art batching techniques. To our knowledge, this is the first study where CI-Skip rules are used in combination with commit batching techniques.

4.2 Data Set

To perform an appropriate comparison of our study against those by Bavand et al. [4] and Abdalkareem et al. [2], our data selection approach and data processing operations are inspired by both of these studies.

Following a prior study by Beheshtian et al. [5], Bavand et al. extracted the top 9 projects from TravisTorrent(unknown release date) that have a failure rate less than 20%. They also added to their study 3 additional top projects with a failure rate between 20% and 40% to understand the impact of high failure rate projects on batching. In summary, they studied 12 projects (5 Java and 7 Ruby) with a failure rate between 6.57% and 40.30% of the commits made to their "master" branch. For each project, they also used the first 100 builds to train their Weighted Smoothing algorithm (refer to Section 2.2), leaving all the remaining commits for testing purposes.

Abdalkareem et al. also used TravisTorrent for their study (release 6/12/2016). They focused their study on projects written in Java, filtering their dataset of 1,283 open source projects down to the 393 projects written in Java. By the inherent nature of the TravisTorrent dataset, the projects they examined contained at least 50 builds and at least 10 watchers. After cloning all 393 projects, they evaluated all the commits made in a project after TravisCI was introduced (i.e., all commits made after the .travis.yml file was first added to the project). They were unable to identify the origin of one such project; hence, it was eliminated from the study. In conclusion, they studied 392 projects, written in Java, that contain at least 50 builds.

To perform our study, we investigated the most recently available TravisTorrent [7] data dump (released 25/1/2017), which contained 3,881,992 builds from 948 projects. As much as possible, we combine the data collection processes of both studied projects to perform our empirical study. First, we filtered the data to retain Java-based projects, yielding 401,767 builds from 241 Java projects. The build data of these 241 projects varies from 4 to 46,889 builds. In order to evaluate the effect of our heuristics on large projects with varying failure rates, we then only retained projects with over 2,000 commits, leaving us with a dataset of 53

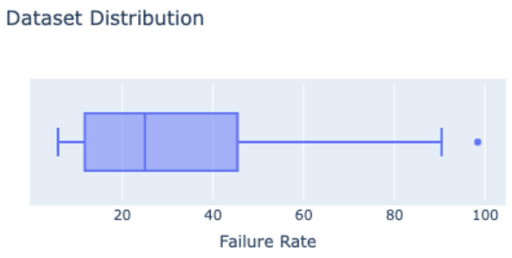


Fig. 4: Distribution of failure rate of 50 projects in the dataset.

projects. Following Bavand et al., we further limited ourselves to builds made only on the “master” branch for each project. Similarly, following their study, we also filtered out the first 100 builds from each project while testing across all heuristics. After doing this, 3 of the 53 projects had to be eliminated since they had less than 100 builds in the master branch to begin with, leaving us with a final dataset of 50 projects with failure rates varying from 5.84% to 98.32%, as shown in Figure 4.

4.3 Variants of Studied Heuristics

We study all LWD batching and the state-of-the-art static and dynamic batching with respect to the same three fallback algorithms, namely BatchBisect, BatchStop4 and BatchDivide4. Hence, both LWD and the state-of-the-art dynamic batching techniques have three variants each – one corresponding to each batching algorithm. Furthermore, each variant of LWD has its sub-variants according to the batch updating technique and factor used (see Table 1). We name our LWD batch updating sub-variant with respect to their factor. For example, Exponential-3 is an exponential batch updating variant used with a factor 3. The variants of static batching, however, are obtained by choosing a fixed batch size. For BatchBisect, batch sizes of 2, 4, 8 and 16 are used in our study, whereas BatchStop4 and BatchDivide4 use batch sizes ≥ 4 (i.e., 4, 8, 16). Each variant of static batching is also named with respect to the batch size used. For example, BatchStop4-8 is used to represent the BatchStop4 algorithm using a batch size of 8 commits.

In RQ3, we also apply CI-Skip rules on incoming commits to remove CI-Skip commits from batches. Algorithm 12 details the detection of CI-Skip commits. The algorithm examines each incoming commit to decide whether it can be excluded from compilation by following the rules of CI-Skip. If the commit cannot be skipped, we append it to the batch of commits scheduled to be built. Once the batch is completed, it is built according to the rules of the batching technique being used and culprit finding techniques are used to detect any failing commit(s). In this paper, we measure the performance of each studied heuristic - LWD and state-of-the-art static and dynamic batching before and after applying CI-Skip.

Table 2: Variants of State-of-the-art Heuristics

Heuristic	Batching Algorithm	Batch Sizes	Variants
Static Batching	BatchBisect	2, 4, 8, 16	*BatchBisect-2, BatchBisect-4, BatchBisect-8, BatchBisect-16*
	BatchStop4	4, 8, 16	*BatchStop4-4, BatchStop4-8, BatchStop4-16*
	BatchDivide4	4, 8, 16	*BatchDivide4-4, BatchDivide4-8, BatchDivide4-16*
Dynamic Batching	All algorithms		*BatchBisect, BatchStop4, BatchDivide4*

Algorithm 12 CI-Skip (Abdalkareem et al. [2])

```
procedure FillBatch: (incomingCommit)
    $i = 0$
    $batch = []$
    $fails = []$
    $batchSize \leftarrow 16$                                  ▷ Default starting batch size
    while $i < batchSize$ do
        if $is\_CISkip(incomingCommit)$ then
            merge incomingCommit without CI build
        else
            $batch.append(incomingCommit)$
            $i \leftarrow i + 1$
        end if
    end while
    $BatchBisect(batch, fails)$                       ▷ or BatchStop4 or BatchDivide4
end procedure
```

4.4 Evaluation Methodology

To address our research questions, we constructed a build simulator (similar to prior work [4, 5, 14, 24]) to sequentially replay the outcomes of the builds of a project. We chronologically group incoming commits into batches, then 'execute' batch builds by iteratively re-calculating the number of builds needed to build each batch. Upon a failing batch build, we then apply one of the bisection-based fallback algorithms (i.e., BatchStop4, BatchBisect and BatchDivide4) to these batches and dynamically update the batch sizes based on the rules of either an LWD variant or CI-Skip rules. For the baselines, we relied on the batching simulator in the replication packages from Beheshtian et al. [5] (static baseline) and Bavand et al. [4] (dynamic baseline) to replicate the results of their techniques on our dataset of 50 projects.

For each fallback algorithm, we ran our 5 LWD variants (and their sub-variants) and all variants of the state-of-the-art static and dynamic baseline batching techniques, then recorded the percentage of builds saved, i.e., the percentage of builds avoided in comparison to building each incoming commit individually (TestAll baseline). To calculate the percentage of builds saved, we use the following formula:

$$builds\_saved = \frac{Number\ of\ builds\ made}{Total\ commits\ in\ the\ project}$$

Here, the number of builds made includes the number of batches built along with all extra builds required in finding failing commits.

To further corroborate our findings, we perform statistical tests with 95% confidence ($\alpha = 0.05$) to identify whether there is a significant difference between the percentage of builds saved by different heuristics. We rely on Wilcoxon signed-rank tests for 2 paired distributions and Friedman Test (with post-hoc Conover tests) for >2 paired distributions to compute statistical differences, then evaluate the magnitude of the differences using the Cliff's Delta effect size and Kendall's W coefficient of concordance, respectively. Following Vargha et al. [28], we interpret Cliff's Delta effect size as: small if $\delta \geq 0.11$; medium if $\delta \geq 0.28$ and large if $\delta \geq 0.43$. Kendall's W coefficient can be interpreted using Cohen's guidelines where $0.1 \leq w < 0.3$ indicates a small effect size, $0.3 \leq w < 0.5$ indicates a moderate effect size and $w \geq 0.5$ indicates a large effect.

To measure and compare the performance of our studied heuristics, we use the performance metric of the median percentage of builds saved by each batching variant across the 50 studied projects. Due to the requirement to fill an entire existing batch before executing a build and bisecting to identify failing commits, batching techniques have a default delay equal to the batch size before the notification of build outcomes to the developer. For this reason, related work on batching techniques [4,5] does not consider notification delay in their performance analysis, in contrast to the work on build outcome prediction.

5 Performance Results of LWD Variants (RQ1)

In this section, we explore the performance of our LWD batching sub-variants and determine the batch updating technique that has the best performance when used with LWD. Overall, LWD techniques can save between a median of 37.38% and 48.86% builds (across the 50 studied projects), as shown in Figure 5. We now discuss these results in more detail.

Picking the right factor value is important for Linear and Random LWD variants, while it matters less or not at all for the other variants.

First, for the Linear Batch Updating technique, we performed a Friedman test with a post-hoc Conover test to determine any statistically significant performance differences in the percentage of builds saved for the variants using factor values 1 to 4. While, for each of the 3 fallback mechanisms, a significantly better factor value was identified (see p-values in Table 3, and the winning factor values in Table 4), the winning value was not consistent. With the BatchDivide4 fallback mechanism, factor 3 was the best, with a Kendall's W effect size statistic of $w = 0.45$, indicating a large effect size. With BatchBisect and BatchStop4, however, no linear variant performed significantly better. However, based on the median percentage of builds saved, factor values 1 and 4, respectively, performed slightly better.

On the other hand, no particular Random variant was performing statistically better in comparison to other variants. However, we noticed that for the BatchBisect fallback algorithm, the Random-Linear variant performed poorly in comparison to other Random variants (Exponential and Jump), based on a Friedman test with post-hoc Conover tests comparing them. The Random-Jump and

Table 3: p-values for for the LWD Batching Updating Variants (Table 4 shows the corresponding best factor value). Bold indicates statistical significance for $\alpha = 0.05$

	Linear	Exponential	Mixed	MFU	Random
Test Used	Friedman	Pairwise Wilcoxon	Pairwise Wilcoxon	Pairwise Wilcoxon	Friedman
Post-Hoc Test	Conover	n/a	n/a	n/a	Conover
BatchBisect	**2.26e-03**	0.91	0.98	0.94	**4.91e-05**
BatchStop4	**2.90e-14**	0.63	0.78	0.84	**0.0032**
BatchDivide4	**1.43e-14**	0.79	0.71	0.79	**1.32e-07**

Table 4: Best factor values for each combination of LWD Batching Updating Variant and fallback algorithm, as well as best LWD variant per fallback mechanism. Bold indicates the factor values that were statistically significantly better than other factor values according to Table 3, while other factor values were determined based on higher median performance values (no significant improvement).

	Linear	Exp.	Mixed	MFU	Random	Best LWD subvariant		
						p-value	Variant	Median %Builds Saved
BatchBisect	**1**	2	2	2	**Jump**	1.13e-08	Linear-1	41.55%
BatchStop4	**4**	2	2	2	**Exp.**	4.43e-11	Linear-4	45.57%
BatchDivide4	**3**	2	2	2	**Exp.**	5.83e-11	Exp.-2	42.35%

Random-Exponential sub-variants performed only slightly better based on their higher median performance for the BatchBisect, BatchStop4 and BatchDivide4 fallback algorithms respectively.

Finally, to identify any statistical differences in the performance between the factor values 2 and 3 for the Exponential, Mixed and MFU techniques, we used Pairwise Wilcoxon tests (see Table 3). However, none of the sub-variants of the MFU, Mixed and Exponential techniques showed any statistical difference, i.e., picking the right factor value does not matter that much for them.

Hence, to determine a factor with better performance for these sub-variants (for use in the remainder of this paper), we inspect the boxplots of the performance distributions (Figure 5) to find factor values with higher median values. We found that Factor 2 is better in this regard than value 3 for Exponential, Mixed and MFU techniques, and select the former value for the remainder of the paper.

Amongst the LWD sub-variants of Table 4, Linear techniques save higher percentage of builds for BatchBisect and BatchStop4 fallback algorithms, while the Exponential technique saves most builds for BatchDivide4.

As reported in Table 4, we picked the sub-variants with the best performance identified previously, then perform one Friedman test per fallback algorithm to identify the most successful LWD sub-variant. Based on post-hoc Conover tests (with Bonferroni correction of $\frac{\alpha}{2}$), all the BatchBisect and BatchDivide4 LWD variants have statistically similar performances. Based on the box plots of Figure 5, we note that Linear-1 BatchBisect and Exponential-2 BatchDivide4 have the higher median performance (we use these variants in the remainder of this

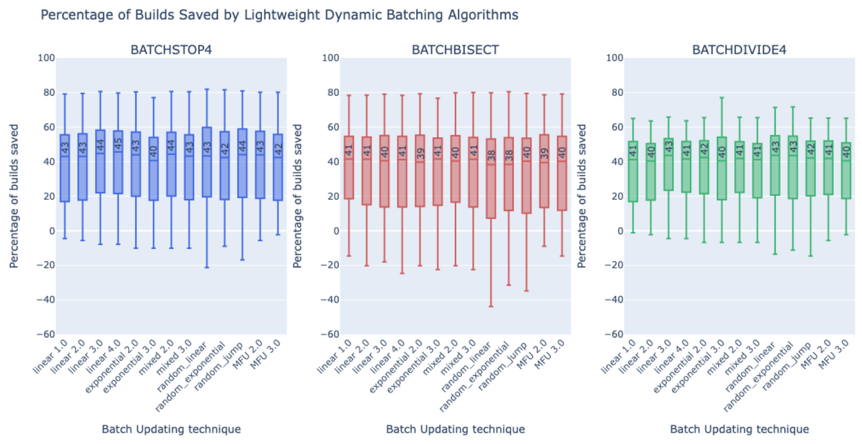


Fig. 5: Performance of all the lightweight batch updating techniques.

paper). Linear-4 BatchStop4 outperformed other BatchStop4 LWD with a large effect size of $w = 0.27$.

The Linear-4 BatchStop4 saves the highest percentage of builds. We perform a final statistical analysis to compare the best techniques of each fallback algorithm, as reported in Table 4, to each other. We find that the Linear-4 BatchStop sub-variant has a significantly higher percentage of saved builds than the Linear-1 BatchBisect and Exponential-2 BatchDivide4 subvariants. It also has higher median (45.57%), maximum (79.68%) build savings along with better skewness, we will use this sub-variant in the remainder of this paper.

Amongst several hypotheses, we believe that the differences in the performances of the LWD variants arise due to the nature of the underlying culprit finding algorithms. According to Beheshtian et al. [5] the best batch sizes used with the BatchBisect algorithm range from 4 to 8, with smaller batch sizes making better savings. From our analysis of LWD, we revealed findings similar to these recorded observations from literature. We found the BatchBisect algorithm to have worked best with Linear-1 LWD, whose small factor sizes allow smaller increases and decreases in the batch size. If at any point in CI, the fallback limit was reached and the batch size was reset to 1, Linear-1 LWD would continue to generate smaller batch sizes. While Exponential, Mixed and MFU LWD used factor size 2 to provide maximum savings with BatchBisect, Table 4 shows that their performance was statistically similar to that of using factor size 3.

On the other hand, Beheshtian et al. also found that bisection is not effective for batch sizes of 4 and less. Hence, by stopping bisection related builds after reaching the batch size 4, BatchStop4 can improve over BatchBisect by 2.69%. Furthermore, Bavand et al. [4], showed that the BatchDivide4 technique used larger batch sizes on projects with low failure rates using their Dynamic Batching algorithm, to make the highest savings amongst their studied algorithms. Larger factor sizes of 4 and 3 can make larger changes in batch sizes, enabling our Linear-4 to achieve larger batch sizes faster than Linear-1 can. Similarly, by using multiplication the

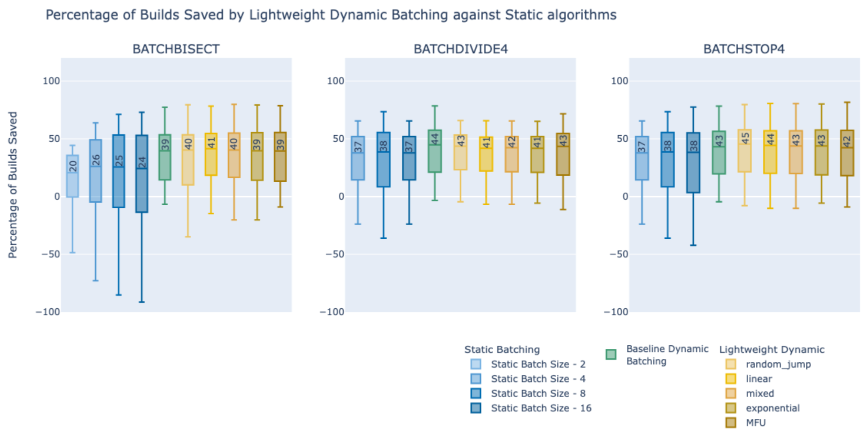


Fig. 6: Comparison of the performance of the best incarnation of our LWD variants against Baseline Dynamic Batching and all variants of Static Batching techniques

Exponential variant can increase batch sizes more rapidly allowing BatchDivide4 to reach its ideal batch size more quickly.

Summary of RQ1: The Linear-4 BatchStop4 LWD variant perform the best of all LWD variants, either statistically significantly or in terms of higher median percentages of saved builds.

6 Comparing LWD variants with baseline algorithms (RQ2)

This section compares our the best of our LWD variants against the state-of-the-art static and dynamic batching techniques.

Our LWD variants save a median of 4.75% more builds than the static baseline technique (significant difference).

Figure 6 compares the performance of the best incarnation of our LWD variants (as listed in Table 4) against all variants of the static batching technique. We found that static batching techniques can save a median of 20.73% up to 38.71% builds across the 50 studied projects, while lightweight dynamic batching can save a median of 38.23% to 45.57% of builds.

By analyzing the median of the performance differences between each static batching variant against its corresponding LWD variants (e.g., Static BatchBisect versus Linear-X BatchBisect, Exponential-X BatchBisect etc.), we found that LWD consistently outperforms the static batching techniques for each fallback algorithm. LWD BatchBisect variants save a median 10.71% more builds in comparison to baseline static BatchBisect, while LWD BatchStop4 saves a median of 4.75% more builds in comparison to baseline static BatchStop4, and LWD BatchDivide4 saves 1.73% more builds than Baseline BatchDivide4.

Corresponding Friedman tests (Table 5) between each baseline variant and corresponding LWD variants (using the optimal factor values established in Ta-

Table 5: p-values comparing Static Batching Variants (columns) against LWD Batching Variants (rows) (using the corresponding best factor value shown in Table 4). Bold indicates statistical significance for $\alpha = 0.05$.

	BatchBisect				BatchStop4			BatchDivide4		
	2	4	8	16	4	8	16	4	8	16
Linear	**< 2e-16**	**1.0e-14**	**1.6e-11**	**1.8e-13**	**6.1e-16**	**2.1e-11**	**2.8e-14**	**3.7e-08**	**0.0036**	**5.8e-08**
Exponential	**1.8e-12**	**3.4e-09**	**1.4e-06**	**3.6e-08**	**4.9e-09**	**1.8e-05**	**1.1e-07**	0.0719	1.0	0.0934
Mixed	**< 2e-16**	**7.9e-15**	**1.3e-11**	**1.4e-13**	**2.8e-10**	**1.7e-06**	**7.1e-09**	**0.0107**	1.0	**0.0144**
Random	**1.4e-09**	**1.2e-06**	**2.1e-04**	**9.3e-06**	**0.0077**	0.8531	0.0501	**2.0e-08**	**0.0024**	**3.3e-08**
MFU	**3.7e-11**	**5.0e-08**	**1.5e-05**	**4.8e-07**	**3.6e-07**	**0.0005**	**5.8e-06**	1.0	1.0	1.0

ble 4), found statistically significant differences (in favour of LWD) for $\frac{41}{50}$ of the studied pairs of heuristics (static approach, LWD), with effect sizes ranging from 0.23 to 0.61 (i.e., large). However, as shown in the last three columns of Table 5, we did not find any statistical difference between the BatchDivide4 MFU-2 and BatchDivide4 Exponential-2 LWD sub-variant the Static BatchDivide4 variants. Similarly, we did not find any statistical difference between the baseline static BatchStop4-8/BatchStop4-16 variants and our LWD BatchStop4 Random Exponential sub-variant (shown in row 4 of Table 5).

On the other hand, we calculated a significant difference in $\frac{8}{10}$ pairs having a variant of random batch updates compared to static batching. The magnitude of this difference was found to range from 0.20 to 0.37 (large). This observation indicates that a predetermined batch size is not essential for the best performance of batching techniques, because using random batch sizes can be equally effective.

The LWD variants perform equally well as the baseline dynamic batching technique.

Our LWD variants are able to match the performance of the state-of-the-art dynamic batching technique, despite leveraging less resources (by not requiring an offline simulation or repeated calculations of weighted failure rates), all the while adding the benefit of customizability (i.e., tweaking of factor values).

Figure 6 compares the LWD variants against the baseline dynamic technique for all three fallback algorithms. The performance differences between the two families of heuristics vary from 0 to 4% median builds saved across all algorithms. 17 out of the 39 LWD variants (i.e., all Linear, Exponential, Mixed, Random and MFU variants for all three fallback algorithms) have a higher median performance than their corresponding baseline dynamic technique.

When using a Pairwise Wilcoxon test to compare each corresponding pair of LWD variant and baseline dynamic technique in Table 6, we found that although there are some performance differences of up to 4%, they are negligible in the bigger picture. We note no statistical difference (i.e., $p - value < 0.05$) between any LWD variant and the baseline technique for any batching algorithm. This indicates that the performance of the two heuristics are similar to each other.

Table 6: p-values of Wilcoxon Tests comparing LWD Batching Variants (rows) against Baseline Dynamic Batching Variants (columns). (Table 4 shows the corresponding best factor value). Bold indicates statistical significance for $\alpha = 0.05$.

Algorithm	BatchBisect	BatchDivide4	BatchStop4
Linear	0.7355	0.4119	0.7944
Exponential	0.9923	0.3154	0.9769
Mixed	0.8243	0.3201	0.9692
MFU	0.9923	0.2796	0.9846
Random	0.8168	0.481	0.8018

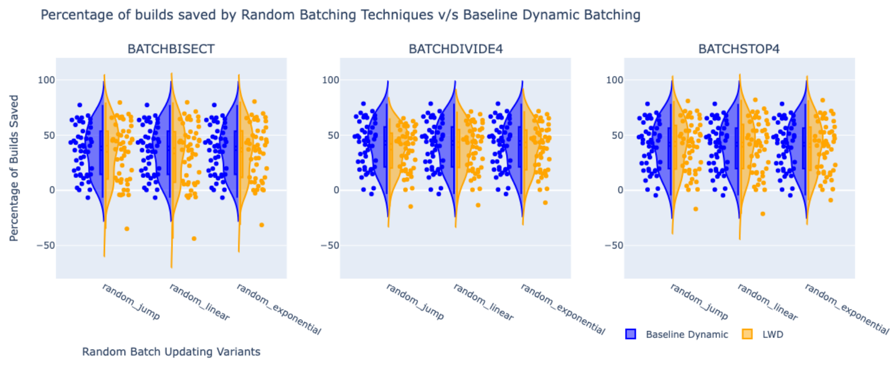


Fig. 7: Comparing the performance of Random Batch Size Updating variants against the baseline Dynamic Batching techniques

These statistical similarities indicate that the use of simpler LWD variants, some even using random batch size updating techniques, is equally effective as using the state-of-the-art dynamic batching technique. Figure 7 compares the random batch updating variants (i.e., Random Linear, Random Exponential and Random Jump) to the baseline dynamic technique, showing that, with the exception of some outliers, data points of the two techniques are distributed similarly across the nine plots.

Summary of RQ2: *The LWD variants save a median of 4.44% to 10.71% more builds than baseline static algorithms. Furthermore, the LWD variants (including the random sub-variants) perform equally well as more complex state-of-the-art baseline dynamic batching techniques.*

7 Performance results of CI-Skip rule integration (RQ3)

Motivated by the success of the lightweight dynamic batching (LWD) rules, we also evaluate the combination of LWD variants with another lightweight rule-based technique, i.e., CI-Skip rules [2]. Algorithm 12 details how CI-Skip rules were integrated with batching techniques.

Integrating CI-Skip rules with dynamic batching techniques does not correlate with significant build savings.

Figure 8a compares the performance of the LWD technique before and after applying CI-Skip rules. From the figure, we see that although the median percentage of builds saved without using CI-Skip is a median of 0.92% higher than with CI-Skip, the maximum percentage of builds saved is increased or remains unchanged. Similarly, except for MFU, CI-Skip can also improve the minimum percentage of builds saved for LWD.

Using Pairwise Wilcoxon tests to evaluate all pairs of CI-Skip vs. non-CI-Skip versions of each LWD variant, we found no statistical differences when combining CI-Skip rules with LWD, indicating that CI-Skip does not have a significant influence on build savings when used in combination with batching.

Similarly, Figures 8b and 8c compare the performance of the Baseline Static (batch size=16) with and without CI-Skip rules, showing that in those cases as well CI-Skip rules only are able to save an additional 0.87% of builds compared to the non-CI-Skip variant. On the other hand, for Baseline Dynamic batching algorithms we can save a median 0.64% more builds without using CI-Skip. We could find no statistical differences between any two variants with and without CI-Skip for these heuristics.

In contrast to LWD's 41.94% median build savings, using CI-Skip rules by themselves helped to save a median of 5.51% builds across the 50 studied projects. The combination of batching with CI-Skip only saves a median of 0.87% builds in addition. Since CI-Skip rules help to filter out simple commits that are highly likely to yield successful builds, the batching heuristic instead has a higher impact on failing commits than on successful commits.

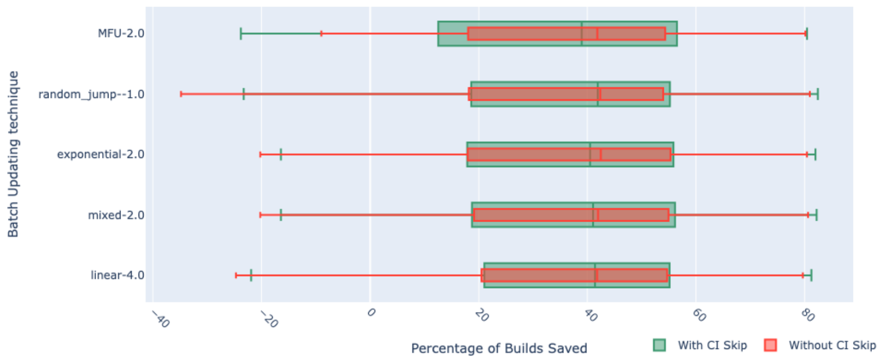
Summary of RQ3: *The combination of LWD and lightweight CI-Skip rules does not correlate with further build savings.*

8 Discussion and Threats to Validity

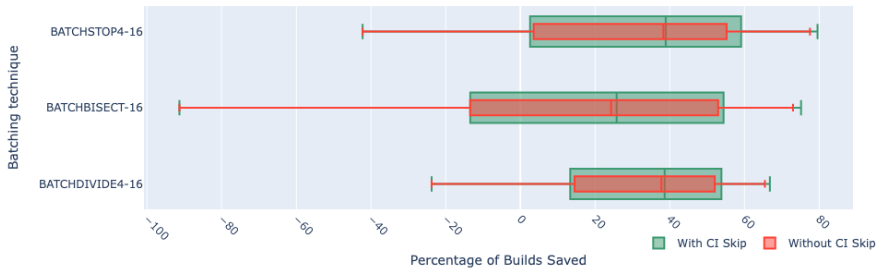
8.1 Wall-clock Time saved by Batching Techniques

Build avoidance heuristics introduced in literature are motivated by the drive to reduce resources used during CI, including cost, energy and time. Up until now, we follow Beheshtian et al. [5], Najafi et al. [24] and Bavand et al. [4] and measure the performance of our studied algorithms in terms of percentage of builds saved. In this section, by using some assumptions, we discuss the median wall-clock time saved by each heuristic. First, for TestAll, i.e., the baseline technique not using any batching techniques, the time required is measured as the sum of the duration of builds of all individual CI commits. Second, we estimate the duration of a batch build to be the time required to build only the last commit of the batch. If the batch build fails, the time required to build the subsequent bisected batches are also added to the build duration. The time saved by a heuristic is then computed by the time required to build each batch in the project using the heuristic. Out of the 50 projects examined, 9 did not record the build duration for their commits. Hence, these were eliminated from our analysis. Figure 9 shows the median time saved by all evaluated batching algorithms.

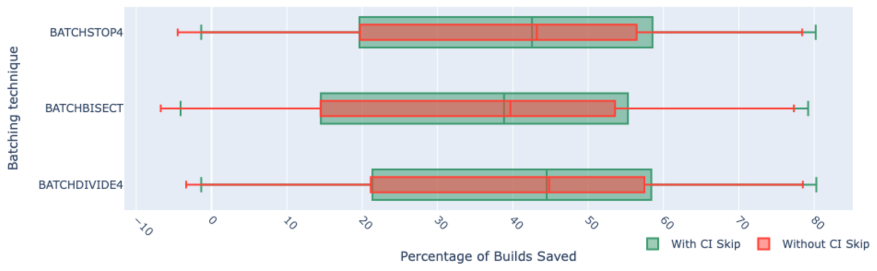
Of the 41 remaining projects, we find that the Baseline Dynamic batching algorithm saves most median time, however this difference is



(a) Lightweight Dynamic Batching



(b) Baseline Static Batching



(c) Baseline Dynamic Batching

Fig. 8: Comparing the performance of all Batching techniques with and without CI-Skip rules

not statistically significant.
Using the BatchBisect culprit finding algorithm, it saved a median 5.49% and 12.49% of build time over LWD and Baseline Static batching respectively. Similarly, it saved a median 5.34% and 10.57% of build time over LWD using BatchStop4 and BatchDivide4 algorithms, respectively, and a median 8.75% and 7.38% time over Static batching algorithms using the same two algorithms. We also find

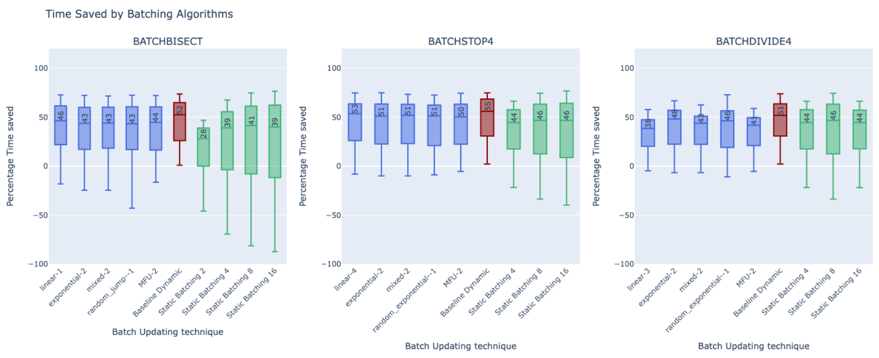


Fig. 9: Distribution of build time saved by all Batching Algorithms.

Table 7: p-values of Wilcoxon Tests comparing Time Saved by LWD Batching Variants (rows) against Baseline Dynamic Batching Variants (columns). (Table 4 shows the corresponding best factor value). Bold indicates statistical significance for $\alpha = 0.05$.

Algorithm	BatchBisect	BatchStop4	BatchDivide4
Linear	0.8175	0.8275	0.5294
Exponential	0.7681	0.7976	0.7878
Mixed	0.7779	0.7878	0.6535
MFU	0.7583	0.7583	0.5723
Random	0.7292	0.7292	0.7196

Table 8: p-values comparing Time Saved by Static Batching Variants (columns) against LWD Batching Variants (rows) (using the corresponding best factor value shown in Table 4). Bold indicates statistical significance for $\alpha = 0.05$.

	BatchBisect				BatchStop4			BatchDivide4		
	2	4	8	16	4	8	16	4	8	16
Linear	**5.7e-09**	**9.0e-05**	**0.0261**	**5.0e-04**	**7.3e-07**	**0.0012**	**1.0e-05**	**0.0152**	**0.0032**	0.1714
Exponential	**5.0e-08**	**6.5e-04**	0.0745	**3.1e-03**	**2.3e-06**	**0.0008**	**2.1e-05**	0.464	1.0	0.055
Mixed	**3.5e-08**	**5.0e-04**	0.0745	**0.0015**	**4.4e-05**	**0.0092**	**3.4e-054**	0.79	0.19	1.0
Random	**4.0e-07**	**0.0040**	0.2756	**0.0106**	**0.0093**	0.8986	**0.0207**	1.0	0.42	1.0
MFU	**2.4e-08**	**0.0005**	0.0609	**0.0024**	**1.2e-04**	**0.0343**	**3.4e-04**	0.055	**0.017**	0.464

that Baseline Static batching saves the least time while using BatchBisect, and saves similar or more time with respect to LWD while using BatchStop4 and BatchDivide4.

On the other hand, when compared to Baseline Dynamic batching algorithms, we see that the LWD algorithm performs statistically similarly when compared to the Baseline Dynamic variants. As the duration of a build across the commits of a given project only has a limited variance over time, the observations about build duration echo those of RQ2, which showed no statistical difference in the percentage of builds saved either.

Tables 7 and 8 discuss the statistical significance of the difference in time saved by our best LWD variants against the Baseline Static and Dynamic algorithms. Based on the findings of the statistical tests along with the boxplots of Figure 9, we see that for 33 out of the 50 pairs comparing the variants of LWD and Static batching, LWD saves significantly more time than Static Batching. Notably, Static BatchDivide4 and Exponential, Mixed and Random LWD have no significant differences in the time they save. Static BatchBisect-8 also saves similar or slightly less time than LWD.

8.2 On Occam's Razor

Through our case studies, we find that the performance of LWD is statistically similar to that of baseline dynamic batching, i.e., the two techniques are equivalent in their performance. However, we argue for the effectiveness of Lightweight Dynamic Batching, keeping in mind the principles of Occam's Razor, which stresses the importance of simplicity when measuring two competing theories [29].

While effective, the baseline dynamic batching algorithm requires calculation of the weighted failure rate of the last 100 commits at every step. Furthermore, it also requires an offline simulator that precomputes ideal batch sizes to be used for corresponding weighted failure rates. On the other hand, the Lightweight Dynamic Batching technique only needs to know the failure rate of the most recent batch build to decide the next batch size. LWD also does not require any offline simulation, since required batch sizes are computed on the go, with minimal effort.

LWD is also a scalable technique that can be applied to a project without prior setup. It can be applied from the first commit of a new project, whereas the baseline dynamic batching algorithm can only be applied after the build outcome of the first 'N' commits (where $N = 100, 200, etc.$) is known.

With straightforward batch size computations, build engineers can understand batching decisions taken by LWD, more easily than that of a blackbox simulator. Moreover, LWD provides the freedom to decide on-the-fly how frequently engineers want to deliver build outcomes to developers. For instance, the number of required builds can be reduced by configuring larger factor values and/or by using techniques like Exponential LWD, which can increment batch sizes by large values and decrement conservatively or vice-versa. Similarly, smaller increments and decrements to batch sizes can be made by using smaller factor values and using techniques like Linear LWD.

On the other hand, where LWD saves significantly higher builds than Static batching algorithms, it does not guarantee a lower wall-clock time of scheduled builds. With Static BatchStop4 and Static BatchDivide4 required between 123%-148% builds in Figure 6, Figure 9 shows that they require only up to 33.99% more time than TestAll. In circumstances where quick delivery of build results is more important than reducing energy consumed by builds, this trade-off between LWD and Baseline batching algorithms can be useful to prioritize the current needs of the team.

8.3 Threats to Validity

We discuss the major threats to the study validity in three categories: construct, internal, and external validity.

Construct Validity: Following the studies by Najafi et al. [24], Bavand et al. [4], and Beheshtian et al. [5], we measure the performance of all studied heuristics in terms of the percentage of builds they save, in comparison to individually building 100% of incoming commits. Although useful, this metric does not give an accurate representation of the time, computing resources or energy conserved by the studied heuristics.

In Section 8.1, we measure the wall-clock time saved by each of our studied heuristics. However, the dataset did not record build durations for many commits for 9 of the 50 projects in our experiment. We attempt to approximate the time saved by evaluating the remaining 41 projects. Additionally, to calculate the time saved by LWD in comparison to baseline algorithms, we estimate the time required to build each batch to be the time required to build the last commit of the batch independently along with time required to bisect the batch to find failing commits. While not 100% accurate, these approximations provide a good idea of the actual wall-clock time differences.

Internal Validity: In our study, all simulation scripts, data analysis and processing was conducted by a single author. This could pose a threat of human error and bias due to faulty scripts or missing data. To minimize this risk, we carefully use the replication packages by Bavand et al. [4], Beheshtian et al. [5] and Abdalkareem et al. [2].

External Validity: In our work, we examine the impact of CI-Skip rules presented by Abdalkareem et al. [2] on commit batching techniques. These CI-Skip rules are designed for Java-based projects, and also led to us choosing Java projects for our study. Furthermore, we also limit ourselves to large, open-source projects containing at least 2,000 commits before 25-01-2017. However, we minimize the threat this poses by incorporating projects with varying failure rates in our study to generalize our results as much as possible. Further analysis will be needed to estimate the results of our proposed LWD heuristics on smaller projects as well as on projects using different programming languages, build and CI tools. The results may also vary for proprietary projects, which will have to be analysed separately.

Conclusion Validity: We study statistical differences between our studied heuristics (LWD, state-of-the-art static and dynamic batching techniques). We use Wilcoxon pairwise signed-rank test when two groups are compared to each other and use pair-wise Friedman tests with the post-hoc Conover test in situations where more than 2 groups are involved. We also use Bonferroni correction wherever applicable to reduce the risk of Type-I errors.

9 Conclusion

Researchers continue to investigate techniques that help to combat the high cost of CI. Existing heuristics rely on build and test reduction and used techniques that involved machine learning or rule based models. In our study, we mainly investigate the impact of simple, rule-based heuristics and commit batching to reduce the number of builds required to run during CI.

Existing state-of-the-art static and dynamic commit batching techniques emphasize on either using the 'ideal' batch size or using more complex mathematical models to dynamically update batch sizes throughout the CI process. On the other hand, we examined the effect of easily customizable 'lightweight techniques' that range from simple arithmetic operations to random and conditional operations to dynamically update the batch size used. We find that our proposed techniques perform better than state-of-the-art static batching algorithms and have save almost similar percentage of builds to the existing, more complex dynamic batching algorithm. However, we also find that while our algorithm uses lesser build time than Static batching algorithm, the time saved by state-of-the-art Dynamic batching is higher, particularly while using the BatchBisect and BatchDivide4 culprit finding algorithms.

Amidst the era and boom of AI, our study strives to show that not every question needs to be solved using intelligent methods. However, we understand that the problem of CI build outcome prediction or batch size estimation requires many factors to be considered, and is too random to predict. Any intelligent solution must either accept a reasonable degree of prediction errors or perform better than random techniques. In such CI problems, simple, pragmatic solutions can be equally if not more effective solutions.

10 Conflict of Interest

All authors declare that they have no conflicts of interest.

11 Data Availability

The datasets generated during and/or analysed during the current study are available in the online replication package [19]

Acknowledgements We would like to thank Dr. Emad Fallahzadeh for his helpful discussions and his feedback on the implementations of the various dynamic batching approaches.

References

1. Abdalkareem, R., Mujahid, S., Shihab, E.: A machine learning approach to improve the detection of ci skip commits. IEEE Transactions on Software Engineering **47**(12), 2740–2754 (2020)
2. Abdalkareem, R., Mujahid, S., Shihab, E., Rilling, J.: Which commits can be ci skipped? IEEE Transactions on Software Engineering **47**(3), 448–463 (2019)
3. Basili, V., Rombach, H.: The TAME project: Towards improvement-oriented software environments. IEEE Transactions on Software Engineering (TSE) **14**(6), 758–773 (1988)
4. Bavand, A.H., Rigby, P.C.: Mining historical test failures to dynamically batch tests to save ci resources. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 217–226. IEEE (2021)
5. Beheshtian, M.J., Bavand, A.H., Rigby, P.C.: Software batch testing to save build test resources and to reduce feedback time. IEEE Transactions on Software Engineering **48**(8), 2784–2801 (2021)
6. Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., Marinov, D.: Deflaker: Automatically detecting flaky tests. In: Proceedings of the 40th international conference on software engineering, pp. 433–444 (2018)

7. Beller, M., Gousios, G., Zaidman, A.: Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 447–450. IEEE (2017)
8. Duvall, P.M., Matyas, S., Glover, A.: Continuous integration: improving software quality and reducing risk. Pearson Education (2007)
9. Elbaum, S., Rothermel, G., Penix, J.: Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 235–245 (2014)
10. Elsner, D., Hauer, F., Pretschner, A., Reimer, S.: Empirically evaluating readily available information for regression test optimization in continuous integration. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 491–504 (2021)
11. Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N., Kandula, S.: Cloudbuild: Microsoft's distributed and caching build service. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 11–20 (2016)
12. Fan, Y., Lv, C., Zhang, X., Zhou, G., Zhou, Y.: The utility challenge of privacy-preserving data-sharing in cross-company defect prediction: An empirical study of the cliff&morph algorithm. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 80–90. IEEE (2017)
13. Herzig, K., Greiler, M., Czerwonka, J., Murphy, B.: The art of testing less without sacrificing quality. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 483–493. IEEE (2015)
14. Jin, X., Servant, F.: A cost-efficient approach to building in continuous integration. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 13–25 (2020)
15. Jin, X., Servant, F.: Cibench: a dataset and collection of techniques for build and test selection and prioritization in continuous integration. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 166–167. IEEE (2021)
16. Jin, X., Servant, F.: 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), pp. 213–225. IEEE (2021)
17. Jin, X., Servant, F.: Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration. Journal of Systems and Software **188**, 111292 (2022)
18. Jin, X., Servant, F.: Hybridcisave: A combined build and test selection approach in continuous integration. ACM Transactions on Software Engineering and Methodology **32**(4), 1–39 (2023)
19. Kamath, D.: Replication package (2023). URL https://doi.org/10.5281/zenodo.13760638
20. Kazmi, R., Jawawi, D.N., Mohamad, R., Ghani, I.: Effective regression test case selection: A systematic literature review. ACM Computing Surveys (CSUR) **50**(2), 1–32 (2017)
21. Liang, J., Elbaum, S., Rothermel, G.: Redefining prioritization: continuous prioritization for continuous integration. In: Proceedings of the 40th International Conference on Software Engineering, pp. 688–698 (2018)
22. Maudoux, G., Mens, K.: Correct, efficient, and tailored: The future of build systems. IEEE Software **35**(2), 32–37 (2018)
23. Menzies, T., Butcher, A., Marcus, A., Zimmermann, T., Cok, D.: Local vs. global models for effort estimation and defect prediction. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 343–351. IEEE (2011)
24. Najafi, A., Rigby, P.C., Shang, W.: Bisecting commits and modeling commit risk during testing. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 279–289 (2019)
25. Pinciroli, C., Beltrame, G.: Buzz: An extensible programming language for heterogeneous swarm robotics. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 3794–3800. IEEE (2016)
26. Rausch, T., Hummer, W., Leitner, P., Schulte, S.: An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 345–355. IEEE (2017)

27. Shi, A.W.: Improving regression testing efficiency and reliability via test-suite transformations. Ph.D. thesis (2020)
28. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000)
29. Webb, G.I.: Occam's Razor. In: C. Sammut, G.I. Webb (eds.) Encyclopedia of Machine Learning, pp. 735–735. Springer US, Boston, MA (2010). DOI 10.1007/978-0-387-30164-8_609. URL https://doi.org/10.1007/978-0-387-30164-8_609
30. Zaha, J.M., Dumas, M., Ter Hofstede, A., Barros, A., Decker, G.: Service interaction modeling: Bridging global and local views. In: 2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06), pp. 45–55. IEEE (2006)
31. Ziftci, C., Reardon, J.: Who broke the build? automatically identifying changes that induce test failures in continuous integration at google scale. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 113–122. IEEE (2017)