Smart Build Targets Batching Service at Google

Kaiyuan Wang
Google Inc
kaiyuanw@google.com

Daniel Rall
Google Inc
dlr@google.com

Greg Tener
Google Inc
gtener@google.com

Vijay Gullapalli
Google Inc
vijaysagar@google.com

Xin Huang
Google Inc
xnh@google.com

Ahmed Gad
Google Inc
ahmedgad@google.com

Abstract—Google has a monolithic codebase with tens of millions build targets. Each build target specifies the information that is needed to build a software artifact or run tests. It is common to execute a subset of build targets at each revision and make sure that the change does not break the codebase. Google's build service system uses Bazel to build targets. Bazel takes as input a build that specifies the execution context, flags and build targets to run. The outputs are the build libraries, binaries or test results. To be able to support developer's daily activities, the build service system runs millions of builds per day.

It is a known issue that a build with many targets could run out of the allocated memory or exceed its execution deadline. This is problematic because it reduces the developer's productivity, e.g. code submissions or binary releases. In this paper, we propose a technique that predicts the memory usage and executor occupancy of a build. The technique batches a set of targets such that the build created with those targets does not run out of memory or exceed its deadline. This approach significantly reduces the number of builds that run out of memory or exceed the deadlines, hence improving developer's productivity.

I. INTRODUCTION

Google has a monolithic codebase and it grows rapidly with an average of more than 100,000 code submissions per day. To make sure that the new changes do not break the existing codebase, Google has adopted Continuous Integration (CI) [7], [11]. For each code change, a CI system first uses the build service system [14] to build the affected libraries and binaries. Then, it runs the affected tests to check if the built libraries and binaries are working as intended.

Google's build service system uses Bazel [1] to build libraries and run tests. Bazel takes as input a set of build specification files that declare build targets. We refer to a build target as a target in the rest of this paper. A target specifies what is needed to produce an artifact, such as a library or binary. A test target specifies what is needed to run tests and check if a code change breaks the codebase. Bazel decides how to build a given target based on the target's specification.

As the codebase grows, each change may affect a lot of targets. For example, a code change on a common library like Guava [4] could affect many Java targets. A C++ compiler change could affect all C++ targets. Moreover, the postsubmit service needs to guarantee that all targets in a given change revision compile and pass all tests. The above use cases are common and require Bazel to build up to millions of targets at once. Running a large number of targets in a build execution may cause *out of memory* (*OOM*) errors in Bazel

as it exceeds the memory of a single machine for dependency analysis. Moreover, Google's infrastructure limits the number of executors, e.g. CPU, GPU and TPU, a build can use in parallel at the build execution time. So executing too many targets in a build may cause the build to take too long and exceed its invocation deadline, resulting in *deadline exceeded* (*DE*) errors. In contrary, executing too few targets in a build may use up all the machines allocated to run builds.

In this paper, we propose a build target batching service (*BTBS*) that partitions a large stream of targets into batches and creates a build for each batch of targets such that those builds do not have OOM or DE errors. The technique relies on a memory estimation model and an executor occupancy estimation model. The memory model predicts the peak memory usage of a build. The occupancy model predicts the average executor occupancy of a build. The technique partitions a large stream of targets into target batches such that the build with the flags and each batch of targets consumes a limited memory or executor occupancy. The results show that BTBS generates few OOM and DE builds with 0.08% OOM rate and 0.05% DE rate, which saves a lot of computational resources used for build failure retries.

The paper makes the following contributions:

- It presents the first technique that effectively creates builds from a large stream of targets with the goal of minimizing the number of OOM and DE errors. It is also the first technique that predicts memory and occupancy usages of build executions.
- It demonstrates that the technique is able to reduce the OOM rate to 0.08% and the DE rate to 0.05%. Our past experience shows that BTBS is critical and improves the developer's productivity by reducing build failure retries due to OOM or DE errors.

II. BACKGROUND

A. Bazel

Bazel is an open-source build and test tool and is widely used within Google. It is responsible for transforming source code into libraries, executable binaries, and other artifacts. Bazel takes as input a set of flags and targets that programmers declare in build files. It supports a large number of command line options and these options can affect the way Bazel generates outputs.

TABLE I: Example build flags

Category	Example
Error checking	–check_visibility enables checking if all dependent targets are visible to all targets to build.
Tool flags	–copt specifies the arguments to be passed to the C++ compiler.
Build semantics	–cpu specifies the target CPU architecture to be used for the compilation.
Execution strategy	–jobs specifies a limit on the number of concurrently running executors during build execution.
Output selection	–fuseless_output restricts Bazel to generate intermediate output files in memory.
Platform	–platforms specifies the labels of the platform rules describing the target platforms.
Miscellaneous	–use_action_cache enables Bazel's local action cache.

```
java_library(                 java_library(                 java_test(
  name = "HelloWorld",          name = "HelloWorldTest",       name = "AllTests",
  srcs = ["HelloWorld.java"],   srcs =                         size = "small",
)                                 ["HelloWorldTest.java"],     tags = ["requires-gpu"],
                                deps = [":HelloWorld"],        deps = [":HelloWorldTest"],
                              )                              )
```
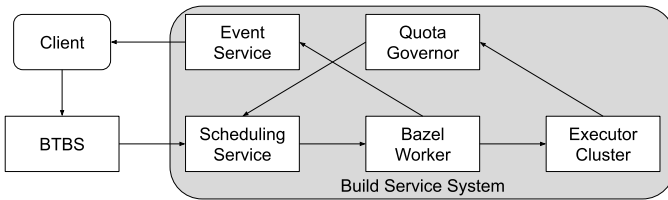
Fig. 1: Example Java targets

Fig. 2: Build service architecture

Table I shows different flag categories with examples. For example, the `–fuseless_output` flag restricts Bazel to generate intermediate output in memory instead of writing it to the disk. The `–jobs` flag specifies a limit on the number of concurrently running executors during a build execution. So `–fuseless_output` and `–jobs` can significantly affect the memory usage and executor occupancy of a build, respectively.

Figure 1 shows some example Java targets. `HelloWorld` is a Java library target that compiles `HelloWorld.java` into a library (`java_library` is the target rule). `HelloWorldTest` is a Java library target that compiles `HelloWorldTest.java` into a library and it depends on the `HelloWorld` target because `HelloWorldTest.java` uses `HelloWorld.java`. `AllTests` is a Java test target that runs the `HelloWorldTest` library using JUnit and the execution requires GPU. When a programmer issues a command to build a target, Bazel first ensures that the required dependencies of the target are built. Then, it builds the desired target from its sources and dependencies. When a programmer issues a command to run a test target, Bazel will first build all dependencies of the test target and then execute the tests.

B. Build Service System

BTBS takes as input a set of flags and a stream of targets, and outputs a set of builds that include all targets with the same flags. The output builds are sent to the build service system [14] for execution. Figure 2 shows how BTBS is connected to the build service system. The system diagram is simplified for brevity.

BTBS splits a stream of targets into batches and creates a build for each batch of targets. Those builds are enqueued to the scheduling service. The scheduling service waits until there are available resources and dequeues a build to a Bazel worker for execution. The worker allocates a fixed amount of memory for each Bazel process and a build runs out of memory if the Bazel process uses more than the allocated memory during execution. Bazel translates the build flags and targets into actions, and sends those actions to the executor cluster for the actual execution. For example, if a test requires GPU then Bazel will send it to the GPU executors. The executor cluster has a large but limited number of executors and each executor talks to a Bazel process to execute actions. The executor cluster also has an action cache to minimize duplicate work. Each Bazel process is configured to use a limited number of executors concurrently to avoid the case where a very large build uses a lot of executors and blocks other build executions. As a consequence, executing a large build that could use more executors than the limit causes the additional actions to queue until some executors become idle again. This may cause the build to exceed its deadline and we call these builds `Type I` DE builds. It is also possible that some actions depend on other actions and they must be executed in sequence. This may cause build to exceed its deadline if some actions are long running. We call these builds `Type II` DE builds. An action uses both memory and executor in the executor cluster. We use an executor service unit (ESU) to unify the expense of both *executor* memory and CPU, and 1 ESU is equal to 2.5GB of memory or 1 executor. Note that the executor memory usage is different from the Bazel memory usage. We refer to the occupancy usage as the ESU used by a build in the rest of the paper. BTBS reduces `Type I` DE errors by limiting the occupancy of a build, and we do not consider `Type II` DE errors because it is not correlated with the build

occupancy usage. BTBS assumes that a build with too much ESU usage is more likely to have queuing actions that cause `Type I` DE errors. The executor cluster reports the executor availability to the quota governor and the scheduling service makes dequeuing decisions based on the executor availability from the quota governor. During a build execution, the Bazel process reports the progress and result to the build event service. Clients of BTBS can use the build request IDs to query the build event service and find the build progress and status in real time.

C. Collateral Damage

Each build may use different executor types. For example, the `AllTests` Java test target can use both x86 CPU and GPU during execution. The build scheduling service throttles builds based on the availability of each executor type. For example, a build that requires both x86 CPU and GPU can only be dequeued when both x86 CPU executors and GPU executors are available. In other words, a build might be delayed if one of its required executor types is unavailable even if all other required executor types are available. Moreover, the delayed build will reserve the executor resources and block other lower priority builds from dequeuing. This design prevents high priority builds from being starved by lower priority builds that use very few ESU [14]. As a consequence, a build that requires more executor types is more likely to be delayed and it may block other lower priority builds from dequeuing.

If a delayed build contains targets that require different sets of executor types, then some of the targets could have been dequeued if they are in a separate build. For example, assume that a build contains one target that only uses x86 executors and another target that uses both x86 and GPU executors. The build could be throttled due to insufficient GPU executors but the target that only uses x86 executors could be dequeued if it is built separately. We call such delays *collateral damage*.

D. Linear Regression

In machine learning, regression analysis is a set of statistical processes for estimating the relationships between a dependent variable (denoted as y) and one or more independent variables (denoted as x). The most common form of regression analysis is linear regression [12]. It tries to find the line that most closely fits the data according to a specific mathematical criterion. For example, the method of ordinary least squares computes the unique line (or hyperplane) that minimizes the sum of squared distances between the true data and that line (or hyperplane).

Given a set of data set $\{y_i, x_{i1}, \ldots, x_{ip}\}_{i=1}^n$ of n statistical units. A simple linear regression model has the following form:

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \mathbf{x}_i^\mathsf{T} \boldsymbol{\beta} + \varepsilon_i, \qquad i = 1, \ldots, n$$

where $^\mathsf{T}$ denotes the transpose, so that $\mathbf{x}_i^\mathsf{T} \boldsymbol{\beta}$ is the inner product between vectors $\mathbf{x}_i$ and $\boldsymbol{\beta}$.

Fitting a linear model to a given data set usually requires estimating the regression coefficients $\boldsymbol{\beta}$ such that the error term $\varepsilon_i = \mathbf{y_i} - \mathbf{x}_i^\mathsf{T} \boldsymbol{\beta}$ is minimized across all n samples. For example,

it is common to use the sum of squared errors $\sum_{i=1}^n \varepsilon_i^2$ as the quality of the fit. Linear models can be efficiently trained using stochastic gradient descent [5].

E. Feature Cross

A feature cross is a synthetic feature formed by multiplying (crossing) two or more features [3]. Crossing combinations of features can provide predictive abilities beyond what those features can provide individually. As a consequence, feature crosses can help us learn non-linear functions using linear regression. A well-known example is that the XOR function $f(x, y)$ where $x, y, f(x, y) \in \{0, 1\}$ is not linearly separable and it cannot be written as $f(x, y) = \alpha x + \beta y + \gamma$ where α, β and γ are real numbers. However, the XOR function can be written as $f(x, y) = x + y - 2xy$, where the xy term is a feature cross for x and y.

In practice, machine learning models seldom cross continuous features. However, machine learning models do frequently cross one-hot feature vectors. Feature crosses of one-hot feature vectors are analogous to logical conjunctions. For example, suppose we bin latitude and longitude, producing separate one-hot five-element feature vectors, e.g. [0, 0, 0, 1, 0]. Further assume that we want to predict the gross income of a person using the binned latitude and longitude as features. Creating a feature cross of these two features results in a 25-element one-hot vector (24 zeroes and 1 one). The single 1 in the cross identifies a particular conjunction of latitude and longitude. By feature crossing these two one-hot vectors, the model can form different conjunctions, which will ultimately produce far better results compared to a linear combination of individual features.

III. BUILD TARGET BATCHING SERVICE

A. EnqueueTargets API

BTBS provides a single remote procedure call (RPC) called `EnqueueTargets`. The RPC is a streaming RPC and it takes as input a sequence of requests and returns to the clients a sequence of responses. The main reason to have a streaming RPC is to avoid needing to stream millions of targets in a single request. We enforce the first `EnqueueTargets` request to include the execution context, build flags and optionally some targets. The execution context points to either a workspace that contains the unsubmitted code, or an existing code revision. The flags and targets are described in Section II-A. The subsequent requests may only contain the remaining targets. BTBS will create a set of builds with the same execution context and flags but different targets. The created builds include all targets in the requests. Each `EnqueueTargets` response contains the enqueued build request ID so that the client can use it to query the build status.

B. Group Targets

Once BTBS receives all targets from the client, it first groups targets by the executor types they use. This avoids the collateral damage as described in Section II-C. For example, all targets that use only x86 executors will be grouped together.

Algorithm 1: Batching targets algorithm

Input: List of targets to batch *allTargets*; Max targets per
    batch *maxTargetsPerBatch*; Memory cutoff value
    memoryCutoff; Occupancy cutoff value
    occupancyCutoff.
Output: Target batches *batches*.

---

batches = []
while *len(allTargets) > 0* **do**
    targets = *allTargets*[:*maxTargetsPerBatch*]
    targets = `limitBatchSizeByCutoff`(*targets,*
     memoryCutoff, MEMORY_MODEL)
    targets = `limitBatchSizeByCutoff`(*targets,*
     occupancyCutoff, OCCUPANCY_MODEL)
    batches.append(*targets*)
    allTargets = *allTargets*[len(*targets*):]
return *batches*
Function `limitBatchSizeByCutoff`(*targets, cutoff,*
modelName)**:**
    low = 0, *high* = len(*targets*) - 1, *cutoffIndex* = 0
    while *low <= high* **do**
       mid = (low + high) / 2
       estimate =
        `getEstimateFromTargets`(*targets[:mid + 1],*
        modelName)
       if *estimate < cutoff* **then**
         cutoffIndex = mid
         low = mid + 1
       else *high = mid - 1*
    return *targets*[:*cutoffIndex* + 1]

All targets that use both x86 and Mac executors will be grouped together. Given a build target, BTBS determines its required executor types by checking the `tags` attribute, e.g. the `requires-gpu` tag in the `AllTests` target in Figure 1. If a target does not have a tag, then BTBS determines the required executor type by the target rule. For example, if a target has the rule `ios_ui_test`, then it uses Mac executors.

Once all targets are grouped by their required executor types, we sort each group of targets in lexicographical order. This is a heuristic to increase the probability that the batched targets share similar dependencies and thus allow Bazel to construct tighter dependency graphs and reduce the memory usage. The idea is that developers tend to declare targets for the same project under the same directory. The targets in the same directory are likely to share dependent targets. For example, target `//a/b/c:t1` may share more dependencies with target `//a/b/c:t2` than target `//d/e/f:t3`. The assumption may not hold for all targets, but this heuristic works well in practice.

C. Batch Targets

BTBS generates a set of target batches for each group of lexicographically sorted targets. Algorithm 1 shows the target batching algorithm. The algorithm takes as input a group of sorted targets *allTargets*, a bound on the max number of targets per batch *maxTargetsPerBatch*, the memory cutoff value *memoryCutoff* above which to consider the build with the batch of targets running out of memory, and the occupancy cutoff value *occupancyCutoff* above which to consider the build with the batch of targets exceeding the deadline. The output *batches* is a list of target batches.

The algorithm first initializes *batches* to an empty list. When *allTargets* is not empty, the algorithm takes the first *maxTargetsPerBatch* targets in *allTargets* as the initial *targets* list for binary search. The first binary search queries the memory model and tries to find the largest sublist of *targets* that fits into the *memoryCutoff* during execution. The second binary search queries the occupancy model and tries to find the largest sublist of *targets* (updated in the first binary search) that uses less than *occupancyCutoff* ESU. The binary searched *targets* list always starts with the same initial target in each iteration. Finally, the batch of *targets* is added to *batches* and removed from *allTargets*.

`limitBatchSizeByCutoff` is the method that does the binary search. It takes as input a list of *targets* to search, a *cutoff* value to fit the final target list and a model name *modelName* to query the machine learning model. The `getEstimateFromTargets` method generates the features from a synthetic build with the same execution context, build flags as present in the first `EnqueueTargets` request, but with a different sublist of *targets*. Then, the method sends the generated features to the model which then returns back an estimate. For the memory model, it returns the estimated peak memory usage of the build. For the occupancy model, it returns the average occupancy of the build. Finally, the `limitBatchSizeByCutoff` method returns a sublist of *targets* that uses less than *cutoff* GB of memory at peak or fewer than *cutoff* ESU on average. Note that the returned *targets* sublist always includes at least one target.

D. Batch Size Reasons

Each target batch is associated with a batch size reason, indicating why the batch of targets is created. We define a target batch to be valid if either (1) it uses less than *memoryCutoff* memory and *occupancyCutoff* occupancy; or (2) it only has one target. Table II shows all possible kinds of batch size reasons. `ONLY_ONE_TARGET` means that the remaining unbatched target list only has one target, and BTBS simply creates a single target batch. `MAX_TARGETS` means that the initial target batch with size *maxTargetsPerBatch* is valid. `ALL_REMAINING_TARGETS` means that the target batch includes all remaining unbatched targets and it is valid. `MAX_MEMORY` means that the target batch is created by a binary search with the memory model and is valid. `MAX_OCCUPANCY` means that the target batch is created by a binary search with the occupancy model and is valid. `MEMORY_ESTIMATE_ERROR` means that the memory model returns an error and the target batch includes a list of targets with a fallback size. `OCCUPANCY_ESTIMATE_ERROR` means that the occupancy model returns an error and the target batch includes a list of targets with a fallback size. These batch size reasons can help us understand the impact of the memory model and occupancy model.

Note that a target batch may have a higher estimate than *memoryCutoff* or *occupancyCutoff* when it only contains a single target. But there is nothing BTBS can do because it cannot split a single target. The memory and occupancy

TABLE II: Batch size reasons

Batch Size Reason	Description
ONLY_ONE_TARGET	Remaining unbatched target list only has one target.
MAX_TARGETS	Initial target batch with size *maxTargetsPerBatch* is valid.
ALL_REMAINING_TARGETS	Target batch includes all remaining unbatched targets and is valid.
MAX_MEMORY	Target batch is created by a binary search with the memory model.
MAX_OCCUPANCY	Target batch is created by a binary search with the occupancy model.
MEMORY_ESTIMATE_ERROR	Target batch includes a fallback size of unbatched targets due to memory error.
OCCUPANCY_ESTIMATE_ERROR	Target batch includes a fallback size of unbatched targets due to occupancy error.

models are served in a separate server and BTBS queries those models via RPC. So it is possible that the RPC fails, e.g. RPC deadline exceeded errors.

E. Create Build

For each target batch, BTBS creates a build with the same execution context and flags as present in the first `EnqueueTargets` request. The build will be enqueued in the build service system [14]. Finally, BTBS sends back all build request IDs generated by the build service system to the clients so that they can use them to track the build statuses. The clients receive a sequence of `EnqueueTargets` responses and each response contains a single build request ID.

F. Build Failure Retry

Even with the memory model and the occupancy model, builds may still fail with OOM or DE errors. If that happens, BTBS can retry those builds on the build service system [14]. For an OOM build, BTBS splits the targets in that build into half and reruns Algorithm 1 for each subset of the split targets. BTBS does not retry single target OOM builds. For a DE build, BTBS reruns Algorithm 1 with the exact same targets in that DE build. The reason is that we expect the build service system to cache some actions done for the DE build, so rerunning the build should be faster and may finish within the deadline.

IV. MEMORY AND OCCUPANCY PREDICTION

A. Data Collection

Bazel is a Java based build system and it runs on JVM. The Bazel process keeps track of the memory and occupancy usage of a build execution.

In terms of the memory usage, Bazel records both the peak heap memory usage and the peak post full GC heap memory usage. The peak heap memory usage is the max heap memory usage during the build execution. Bazel does not necessarily run out of memory if the peak heap memory usage is close to the allocated memory, because JVM can garbage collect (GC) the unused references and free some heap memory. The peak post full GC heap memory usage is the max heap memory usage after a full GC during the build execution. If the peak post full GC heap memory usage is close to the allocated memory, then Bazel will run out of memory because GC cannot free up more memory space. We use the peak post full GC heap memory usage as the predicting label for the memory

model. If no GC is triggered during the build execution, then we use the peak heap memory usage as the predicting label.

In terms of the occupancy usage, Bazel records the exact build execution time and the total executor service time for the build. The build execution time is the wall time of the command-specific execution, excluding Bazel startup time. The executor service time for a Bazel action is the number of executors (or equivalent memory unified by ESU) multiplied by their execution time occupied by that action. The total executor service time for a build is the sum of executor service time for all actions generated by that build. The average executor occupancy is equal to the total executor service time divided by the build execution time. Conceptually, the average executor occupancy measures the average concurrently used executors or memory during a build execution. If the average executor occupancy is above the number of concurrent executors limit, then it is more likely to cause a `Type I` deadline exceeded build. We use the average executor occupancy (in ESU) as the predicting label for the occupancy model.

The build flags and targets are the raw features we use to predict the memory and occupancy usage. The Bazel process is configured to store all execution statistics to a distributed file system [8] and we have a Flume [6] job to generate all features and labels from the execution statistics every day.

B. Feature Engineering

By measuring how important a feature is in predicting the label, we can choose the most representative features instead of all the features. The benefits of feature reduction include (1) speeding up the model training, (2) making the model training stable (training the same model multiple times produces close results), (3) avoiding learning bad features, and (4) reduced data processing, etc. We use the mutual information [10], [13] to directly measure how much information a set of features can provide to predict the label.

Table III shows the set of important features we use to predict both memory and occupancy usages. The build priority does not affect the memory or occupancy usage of a build. But it happens that high priority builds often run a smaller set of targets compared to low priority builds. For example, the number of targets triggered by a human code change is often smaller than the number of targets triggered by a code coverage tool. The Bazel command name is important because it affects the way targets are built. For example, the `test` command not only builds the libraries/binaries but also

TABLE III: Important features

Features	Description
Build priority	The priority of the build.
Command name	The Bazel command name.
Originating user	The username that requests the build.
Product area	The product area under which the build is charged.
Tool tag	The tool that sends targets to BTBS.
Targets	The set of targets in the build.
Target count	The number of distinct targets in the build.
Packages	The set of packages for the corresponding targets in the build.
Package count	The number of distinct packages of all targets in the build.
–cache_test_results	Whether Bazel uses the previously saved test results when running tests.
–cpu	The target CPU architecture to be used for the compilation of binaries during the build.
–discard_analysis_cache	Whether Bazel should discard the analysis cache right before execution starts.
–fuseless_output	Whether to generate intermediate output files in memory.
–jobs	The limit on the number of concurrent executors used during the build execution.
–keep_going	Whether to proceed as much as possible even in the face of errors.
–keep_state_after_build	Whether to keep incremental in-memory state after the build execution.
–runs_per_test	The number of times each test should be executed.
–test_size_filters	Only run test targets with the specified size.
–use_action_cache	Whether to use Bazel's local action cache.

runs the tests. As a comparison, the `build` command only builds a more restricted set, such as libraries and binaries. The originating user and the product area are important because certain users or teams often trigger targets of the same project and their builds often have similar cost. The tool tag is another important feature because the sets of targets triggered by different tools can vary a lot. For example, a presubmit service often triggers a small set of targets given a code change, but the postsubmit service often triggers millions of targets. The targets and the packages, i.e. directories within which the targets are declared, are very important because targets often have different costs to build. Typically, a build with more targets and packages is more expensive than a build with fewer targets and packages. Build flags are also important because they affect how Bazel builds the targets. For example, the `-discard_analysis_cache` flag tells Blaze to discard the analysis cache for the previous build before the next build execution starts, so it affects the memory used by the build. The `-keep_going` flag tells Bazel to proceed even in the face of errors and it may cause a failed build to use more memory compared to exiting the execution once a failure occurs. The `-runs_per_test` flag controls the number of times each test should be executed so it can affect the number of concurrently used executors in the case of parallel execution. The `-test_size_filters` flag filters a subset of test targets to execute and thus can affect both the memory and occupancy usage of a build. In general, the important features selected based on the mutual information indeed affect the memory and occupancy usage, intuitively. There are more important flags but we do not describe them for brevity.

In addition to the basic important features in Table III, we also generate synthetic features to improve the model accuracy. For example, the target and package counts are discretized into quantile buckets. Specifically, we can split the range of target counts by their median, and half of the target counts would fall into the first bucket and the rest half would fall into the second bucket. This strategy converts a numeric feature into a categorical feature. Another example is to split each target path into multiple fragments based on the path delimiter. Specifically, we generate the target prefix splits feature from a target `//a/b/c:t` by splitting its path into `//a/b/c:t`, `//a/b/c`, `//a/b` and `//a`. This helps the model to learn the memory or occupancy usage patterns for targets under different directories.

All basic and synthetic categorical features are used to generate feature crosses (Section II-E). We use an off-the-shelf blackbox optimization tool similar to AutoML [9] to generate a set of candidate feature crosses. The tool trains the model with a small set of steps to see which feature crosses give the best performance. All feature crosses are generated from the basic and synthetic categorical features, and we limit the max number of feature crosses to 6. Finally, the tool returns the best feature crosses for the models. We run feature cross search separately for the memory and occupancy models.

C. Model Training

Once all features are finalized, they will be used to train the memory and occupancy models. We choose to train a regression model because it fits well with the binary search as described in Algorithm 1. Another reason to not train a binary classification model is that the model may perform badly when only few OOM builds are present in the training data.

We want our models to be monotonic over all targets related features. This property makes sure that adding new targets to the build with the same flags always results in an increased memory estimate. If the monotonicity property does not hold,

then the binary search may not work as expected. We achieve the monotonicity property by setting the regularization term to infinity when the weights of the targets related features are negative. Specifically, we use the loss function as follows:

$$L(\boldsymbol{\beta}, \mathbf{x}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \mathbf{x}_i^\mathsf{T} \boldsymbol{\beta}) + R(\boldsymbol{\beta})$$

$$R(\boldsymbol{\beta}) = \sum_{i=1}^{n} r_i(\boldsymbol{\beta_i})$$

$$r_i(\boldsymbol{\beta_i}) = \begin{cases} \boldsymbol{\lambda_i^-} |\boldsymbol{\beta_i}| & \boldsymbol{\beta_i} < 0 \\ \boldsymbol{\lambda_i^+} |\boldsymbol{\beta_i}| & \boldsymbol{\beta_i} \geq 0 \end{cases}$$

$$\boldsymbol{\lambda_i^-}, \boldsymbol{\lambda_i^+} \geq 0$$

where $L(\boldsymbol{\beta}, \mathbf{x}, \mathbf{y})$ is the mean squared error (MSE) loss function and $R(\boldsymbol{\beta})$ is the regularization term. $\boldsymbol{\lambda_i^-}$ and $\boldsymbol{\lambda_i^+}$ are the L1 regularization term for the weight $\boldsymbol{\beta_i}$ when $\boldsymbol{\beta_i} < 0$ and $\boldsymbol{\beta_i} \geq 0$, respectively. We set $\boldsymbol{\lambda_i^-} = \infty$ for the weights of the targets related features. This causes all the weights of the targets related features to be non-negative, thus making the model monotonic over all targets related features.

We train the memory and occupancy models using the last 17 days of all builds that are executed in the build service system. This means that the training and testing datasets include builds that are not created by BTBS. 17 is chosen to cover at least 2 weeks of weekday data and be able to handle holiday weekends. However, using a long period of training data may cause the memory model to slowly capture the new memory patterns of recent builds. For example, it is possible that a memory usage regression is injected into Bazel, which causes otherwise identical builds to use more memory than before. The memory model won't be able to capture the memory usage increase until a couple of days later because a majority of the builds still use less memory. To solve this problem, we train another memory model that uses data from the most recent day and BTBS uses the maximum memory estimate of the two models when performing the binary search. All the models are continuously trained and pushed to production.

V. EVALUATION

A. Production Setup

BTBS is deployed geographically at 3 locations in the US and each location has 5 running jobs that serve traffic all over the world. This distributed setting avoids single point failure. Our experiment lasts from 2020/01/16 to 2020/02/19 (35 days). We report the performance of BTBS using the production data. During the time, BTBS receives 51 million `EnqueueTargets` RPC calls (1.47 million daily on average) and creates 102 million builds (2.92 million daily on average).

The *maxTargetsPerBatch* is set to 900 in Algorithm 1. The memory cutoff value is set to 7GB for high priority builds, 9GB for medium priority builds and 10GB for low priority builds. The reason is that the memory model is not 100% accurate and the number of OOM builds will increase if we

set the cutoff close to the allocated memory (13GB). We set a lower memory cutoff for high priority builds because we cannot afford the human users to wait for OOM failure retries. In contrast, we can tolerate more low priority OOM builds, e.g. builds that collect code coverage and do not block developers. The occupancy cutoff value is set to 500 ESU which is smaller than the max allowed concurrent executors limit (600). If the memory or occupancy model returns errors, then we fallback to a default batch size of 300.

All regularization terms $\boldsymbol{\lambda_i^+}, i = 1...n$ for positive weights are set to 7000. All the learning parameters, e.g. learning rate and batch size, are set to the default values. During the experiment time, the build service system ran 17.8 millions builds on average each day and 34.8% builds were Bazel queries that do not use much memory or any executor. So we excluded those builds and used the rest 11.6 million builds as the training examples daily.

In the rest of this section, 1k represents 1 thousand and 1m represents 1 million. The error is calculated as the difference between the actual and estimated memory usage, i.e. $(y - \hat{y})$.

B. BTBS Performance

Table IV shows some metrics to measure the performance of BTBS. *QPS* and *Latency* show the queries per second and latency in milliseconds of the `EnqueueTargets` streaming RPC over the past 24 hours during the entire experiment span. *StreamTC* shows the total target count per RPC to BTBS. *StreamBC* shows the total generated build count per RPC to BTBS. *BuildTC*, *ExecT*, *Memory* and *Occupancy* show the target count, execution time in seconds, memory usage in gigabytes and occupancy usage in ESU of individual builds created by BTBS.

BTBS is used heavily in production and the average QPS over the past 24 hours ranges from 6 to 27. The min and max QPS over the past 24 hours range from 0 to 12 and from 18 to 89, respectively. BTBS receives less traffic on weekends off the peak hours and more traffic on weekdays during the peak hours. The batching algorithm is efficient and the average latency over the past 24 hours ranges from 265 to 414 milliseconds. The min and max latency over the past 24 hours range from 1 to 81 milliseconds and from 69k to 886k milliseconds, respectively. The latency of BTBS is very small if the clients enqueue less than hundreds of targets, but it could go up to minutes if the clients enqueue millions of targets. The average, min, median and max number of targets received by BTBS per RPC are 674, 1, 2 and 70m, respectively. A majority of the `EnqueueTargets` requests only contain 1-2 targets and they are sent by tools like presubmit failure rerunner, culprit finder, flaky test detector, etc. The postsubmit service could send millions of targets to BTBS because it needs to make sure that all tests can pass at a given revision. The average, min, median and max number of builds created by BTBS per RPC are 2, 1, 1 and 84k, respectively. This is consistent with the number of targets received by BTBS because BTBS creates more builds as it receives more targets per RPC. Since a majority of the `EnqueueTargets` requests

TABLE IV: BTBS performance

Metrics	Avg	Min	Median	Max
QPS	6-27	0-12	4-24	18-89
Latency	265-414	1-81	150-179	69k-886k
StreamTC	674	1	2	70m
StreamBC	2	1	1	84k
BuildTC	339	1	24	900
ExecT	394	0.1	250	7923
Memory	3.87	0.04	2.82	13.26
Occupancy	18.3	0	2.8	2066.4

only contain 1-2 targets, BTBS only needs to create a single build for most of the time. The average, min, median and max number of targets per build created by BTBS are 339, 1, 24 and 900, respectively. Note that the number of targets per build is bounded by the *maxTargetsPerBatch*. The average, min, median and max execution time of builds created by BTBS are 394, 0.1, 250 and 7923 seconds, respectively. It indicates that most builds run with a couple of seconds to minutes but some builds can run with 1-2 hours. A majority of builds have a deadline of 1.5 hours and will be expired if they do not finish by the deadline. The average, min, median and max memory usage of builds created by BTBS are 3.87, 0.04, 2.82 and 13.26 GB, respectively. It shows that many builds take less than a few GB of memory but some of them can still use up all the allocated memory. The average, min, median and max occupancy of builds created by BTBS are 18.3, 0, 2.8 and 2066.4 ESU, respectively. The max occupancy usage is greater than the max allowed concurrent executors because some builds use more executor memory. It shows that a majority of builds only use a few concurrent executors or executor memory but some builds can use thousands ESU. It is worth mentioning that some builds hit the cached actions so they have 0 occupancy usage.

C. Model Impact

Table V shows the build count distribution by batch size reasons. *#Build* shows the number of builds. *#OOM(%)* shows the number of OOM builds and its percentage out of all builds in each batch size reason. *#DE(%)* shows the number of DE builds and its percentage out of all builds in each batch size reason. *%Build* shows the percentage of builds in each batch size reason out of all builds. *%OOM* shows the percentage of OOM builds in each batch size reason out of all OOM builds. *%DE* shows the percentage of DE builds in each batch size reason out of all DE builds.

Builds with batch size reasons `ONLY_ONE_TARGET`, `MAX_TARGETS`, `ALL_REMAINING_TARGETS`, `MAX_MEMORY`, `MAX_OCCUPANCY`, `MEMORY_ESTIMATE_ERROR` and `OCCUPANCY_ESTIMATE_ERROR` account for 21.12%, 32.95%, 33.06%, 12.71%, 0.06%, 0.10% and 0.00% of the total builds, respectively. The memory model is used in all builds with batch size reasons `MAX_TARGETS`, `ALL_REMAINING_TARGETS`, `MAX_MEMORY`, `MAX_OCCUPANCY` and `OCCUPANCY_ESTIMATE_ERROR`, which is 78.78% of all builds. The occupancy model is used in all builds with batch size reasons `MAX_TARGETS`, `ALL_REMAINING_TARGETS`,

`MAX_MEMORY`, `MAX_OCCUPANCY`, which is 78.78% of all builds. `ONLY_ONE_TARGET` builds use neither the memory model nor the occupancy model because BTBS always creates a build if a single target is left in the remaining target list. `MEMORY_ESTIMATE_ERROR` and `OCCUPANCY_ESTIMATE_ERROR` builds may use the memory and occupancy models, respectively, in the initial binary search iterations before the failure. So both the memory and occupancy models are heavily used in BTBS.

`MAX_MEMORY` builds provide an indicator on the potential number of OOM builds if BTBS simply creates builds with *maxTargetsPerBatch* targets. It shows that 0.36% of `MAX_MEMORY` builds still run out of memory and they account for 58.75% of all out of memory builds. This is expected because the `MAX_MEMORY` builds should use more memory than other builds. The `MEMORY_ESTIMATE_ERROR` builds approximately indicate how BTBS would behave without the memory model. It shows that 0.93% `MEMORY_ESTIMATE_ERROR` builds run out of memory. If the `MAX_MEMORY` builds have the same out of memory rate as the `MEMORY_ESTIMATE_ERROR` builds, we would have 74k more out of memory builds during the experiment. In practice, the memory model is very important because an OOM build is very expensive to retry and it blocks the developer's productivity. The memory model is useful in limiting the memory usage while maximizing the batch density. BTBS used a small *maxTargetsPerBatch* before the memory model existed, the problem was that it generated too many builds and used up almost all Bazel workers, which blocked other builds from running.

`MAX_OCCUPANCY` builds provide an indicator on the potential number of `Type I` DE builds if BTBS simply creates builds with *maxTargetsPerBatch* targets. It shows that 1.4% of `MAX_OCCUPANCY` builds still exceed the deadlines and they account for 1.84% of all deadline exceeded builds. Interestingly, a majority of DE builds are not `MAX_OCCUPANCY` builds. This indicates that most DE builds are `Type II` DE builds. Out of all DE builds, 98.68% builds use less than or equal to 600 ESU and only 1.32% builds use more than 600 ESU. This confirms that `Type II` DE builds dominate in the current setup. Unfortunately, we only have 716 `OCCUPANCY_ESTIMATE_ERROR` builds and no one exceeds the deadline, so it's hard to quantitatively measure how many more deadline exceeded builds we would have without the occupancy model. In practice, we used to have more DE builds before the occupancy model exists, so we believe that the occupancy model is useful to reduce `Type I` DE errors.

Table VI shows the build target count (*BuildTC*), execution time in seconds (*ExecT*), memory usage in GB (*Memory*) and occupancy usage in ESU (*Occup*) on average and median, respectively, for each batch size reason.

The `MAX_MEMORY` builds use 8.1GB and 8.3GB memory on average and median, respectively. This indicates that `MAX_MEMORY` builds use more memory than other builds and the usage is close to the memory cutoff. The `MAX_OCCUPANCY` builds use 471.9 and 521.6 occupancy on average and median, respectively. This indicates that `MAX_OCCUPANCY` builds use

TABLE V: Build count distribution by batch size reasons

Batch Size Reason	#Build	#OOM(%)	#DE(%)	%Build	%OOM	%DE
ONLY_ONE_TARGET	21.6m	22.5k (0.10)	6.6k (0.03)	21.12	28.13	14.22
MAX_TARGETS	33.7m	1.9k (0.01)	6.8k (0.02)	32.95	2.34	14.59
ALL_REMAINING_TARGETS	33.8m	7.6k (0.02)	21.0k (0.06)	33.06	9.52	45.17
MAX_MEMORY	13.0m	46.9k (0.36)	11.2k (0.09)	12.71	58.75	24.13
MAX_OCCUPANCY	59.3k	35 (0.06)	855 (1.4)	0.06	0.04	1.84
MEMORY_ESTIMATE_ERROR	104.3k	967 (0.93)	25 (0.02)	0.10	1.21	0.05
OCCUPANCY_ESTIMATE_ERROR	716	0 (0.00)	0 (0.00)	0.00	0.00	0.00
Sum	102.3m	79.9k (0.08)	46.5k (0.05)	100	100	100

TABLE VI: Build stats by batch size reasons

Batch Size Reason	Avg				Median			
	BuildTC	ExecT	Memory	Occup	BuildTC	ExecT	Memory	Occup
ONLY_ONE_TARGET	1	241	2.2	2.0	1	92	1.6	0.3
MAX_TARGETS	900	371	4.0	33.4	900	275	3.7	18.8
ALL_REMAINING_TARGETS	67	403	3.2	13.2	5	227	2.4	2.5
MAX_MEMORY	154	679	8.1	15.2	9	574	8.3	2.6
MAX_OCCUPANCY	506	1424	5.5	471.9	555	1301	5.6	521.6
MEMORY_ESTIMATE_ERROR	239	443	3.0	21.0	300	306	1.8	11.9
OCCUPANCY_ESTIMATE_ERROR	222	391	2.2	19.6	300	273	1.6	11.3

(a) RMSE by actual memory

(b) Build count by memory error

(c) Build count by actual memory

Fig. 3: Memory model accuracy (GB)

(a) RMSE by actual occupancy

(b) Build count by occupancy error

(c) Build count by actual occupancy
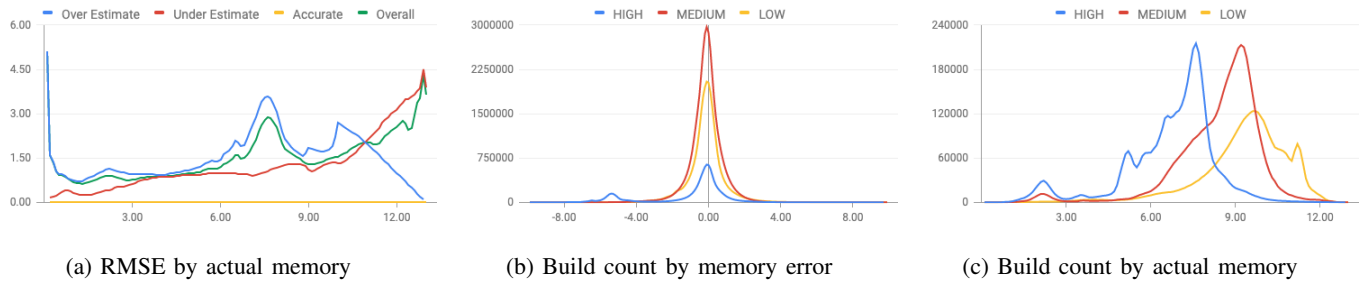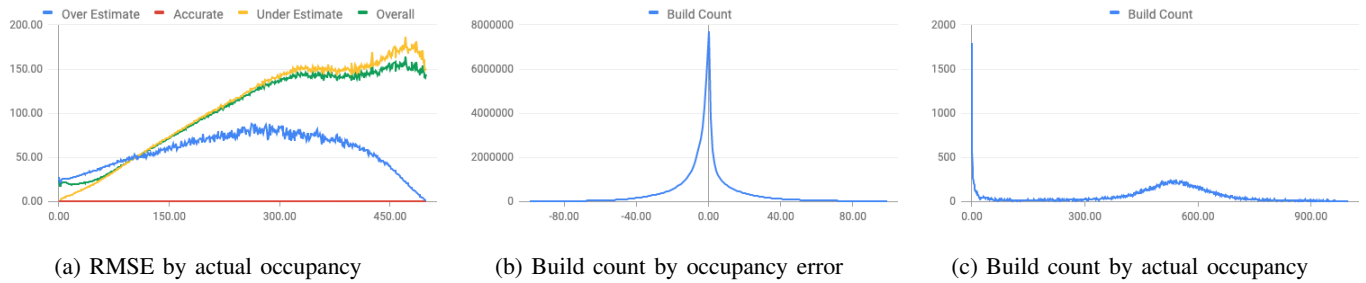
Fig. 4: Occupancy model accuracy (ESU)

more occupancy than other builds and the usage is close to the occupancy cutoff. Moreover, it takes 1424s and 1301s to run MAX_OCCUPANCY builds on average and median, respectively. This indicates that MAX_OCCUPANCY builds are longer running than other builds and the execution time is proportional to the occupancy usage in general. The result shows that the memory and occupancy models can restrict the memory and occupancy usage of the builds, thus reducing the OOM or DE errors.

ONLY_ONE_TARGET builds are single target builds, so they typically have the least execution time and memory/occupancy

usage as expected. All MAX_TARGETS builds have 900 targets and BTBS cannot generate builds with more than *maxTargetsPerBatch* targets. So MAX_TARGETS builds typically have more execution time and memory/occupancy usage. ALL_REMAINING_TARGETS builds are generated at the end of Algorithm 1, so these builds typically have few targets and use little memory and occupancy. The average build target count for the MEMORY_ESTIMATE_ERROR and OCCUPANCY_ESTIMATE_ERROR builds is less than 300 because the initial binary search

iterations may already cut the batch size to be less than 300. Both MEMORY_ESTIMATE_ERROR and OCCUPANCY_ESTIMATE_ERROR builds are rare and they use little memory and occupancy.

D. Model Accuracy

Figure 3a shows the root mean square error (RMSE) of the memory model grouped by different actual memory usages (0GB to 13GB with a step of 0.1GB). The graph is broken down by whether the memory model overestimates (blue line), underestimates (red line) or accurately predicts (yellow line) the actual memory usage, respectively. It also shows the overall RMSE (green line). We can see that the memory model heavily overestimates builds which use very little memory. A common reason is that some targets may finish without GC so their memory usage is recorded much larger than that after the GC. The memory model learns from builds without GC that some targets have large memory usage, so it often overestimates the memory usage of builds with the same targets that finish after GC. The model also overestimates builds which use 7-8GB memory. Many of these builds are single target builds and the model gives large memory estimates for all of these builds. In general, the model tends to have a larger error in overestimation compared to underestimation. The error tends to go up for larger memory builds. Figure 3b shows the number of builds grouped by different error values. The graph is broken down by the build priority. It shows that most builds, regardless of their priorities, have an error close to 0, which means that the model performs well. Figure 3c shows the number of MAX_MEMORY builds grouped by their actual memory usages. We can see that the memory usage of high, medium and low priority builds peaks at around 7GB, 9GB and 10GB, respectively. This is consistent with our setup.

Figure 4a shows the RMSE of the occupancy model grouped by different actual occupancy usages (0 ESU to 500 ESU with a step of 1 ESU). It seems that the occupancy error increases for larger occupancy builds. The overestimation error increases slowly and then decreases until it diminishes at 500. The model only underestimates builds that use more than 500 ESU (not shown in the graph). The underestimation error increases for larger occupancy builds. The overall RMSE increases as the actual build occupancy usage increases. This is expected because most builds use <100 ESU and the model does not have much data to learn larger occupancy builds. Figure 4b shows the number of builds grouped by different occupancy error values. It shows that most builds have an error close to 0, which shows that the model performs well. Figure 4c shows the number of MAX_OCCUPANCY builds grouped by their actual occupancy usages. We can see that the occupancy usage peaks at both 0 and around 500 ESU. The peak at 0 ESU is caused by many builds hitting the action cache in the executor cluster [14] and these builds do not occupy any executor. The peak at around 500 ESU is consistent with our setup.

VI. RELATED WORK

We do not find any related work about using machine learning models to construct builds that use limited resources to reduce OOM and DE errors. This problem is quite common in Google and we expect it to be common in other companies that have a monolithic code repository and many build targets. Our experience is that a service like BTBS becomes more important as the need to build a large number of targets increases. We believe that this work provides an insight for developers who have a similar need, e.g. Bazel [1] and Buck [2] users.

Google's TAP service [11] mentions that the test executions could fail due to OOM errors, but it does not give a solution. BTBS is designed exactly to solve that problem. In fact, the TAP service is one of the BTBS users.

The build service system [14] describes how the build is executed remotely in Google. BTBS is one of the build service system users. It is designed to workaround the constraints, i.e. limited memory allocation and max allowed concurrent executors, imposed by the build service system which is not capable of building a very large number of targets.

VII. CONCLUSIONS

In this paper, we describe the first build target batching service (BTBS) that is able to partition a large stream of targets into a set of target batches such that the builds created from those target batches do not have OOM or DE errors. We discuss the batching algorithm as well as the machine learning models used in BTBS. Overall, we show that BTBS is able to achieve a low OOM rate (0.08%) and a low DE rate (0.05%). We believe that BTBS introduces useful insights and can help in designing new target batching systems.

REFERENCES

[1] Bazel build tool. https://bazel.build/.
[2] Buck build tool. https://buck.build/.
[3] Feature cross. https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture/.
[4] Google guava. https://github.com/google/guava/.
[5] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT*. 2010.
[6] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. *ACM Sigplan Notices*, 2010.
[7] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, 2003.
[9] F. Hutter, L. Kotthoff, and J. Vanschoren. *Automated Machine Learning*. 2019.
[10] A. Kraskov, H. Stögbauer, and P. Grassberger. Estimating mutual information. *Physical review E*, 2004.
[11] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *ICSE-SEIP*, 2017.
[12] D. C. Montgomery, E. A. Peck, and G. G. Vining. *Introduction to linear regression analysis*. 2012.
[13] B. C. Ross. Mutual information between discrete and continuous data sets. *PloS one*, 2014.
[14] K. Wang, G. Tener, V. Gullapalli, X. Huang, A. Gad, and D. Rall. Scalable Build Service System with Smart Scheduling Service. In *ISSTA*, 2020.