BeeCluster: Drone Orchestration via Predictive Optimization

Songtao He
MIT CSAIL
songtao@mit.edu

Favyen Bastani
MIT CSAIL
favyen@csail.mit.edu

Arjun Balasingam
MIT CSAIL
arjunvb@csail.mit.edu

Karthik Gopalakrishna
MIT
karthikg@mit.edu

Ziwen Jiang
MIT CSAIL
ziwenj@csail.mit.edu

Mohammad Alizadeh
MIT CSAIL
alizadeh@csail.mit.edu

Hari Balakrishnan
MIT CSAIL
hari@csail.mit.edu

Michael Cafarella
MIT CSAIL
michjc@csail.mit.edu

Tim Kraska
MIT CSAIL
kraska@csail.mit.edu

Sam Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

The rapid development of small aerial drones has enabled numerous drone-based applications, e.g., geographic mapping, air pollution sensing, and search and rescue. To assist the development of these applications, we propose BeeCluster, a drone orchestration system that manages a fleet of drones. BeeCluster provides a *virtual drone* abstraction that enables developers to express a sequence of geographical sensing tasks, and determines how to map these tasks to the fleet efficiently. BeeCluster's core contribution is *predictive optimization*, in which an inferred model of the future tasks of the application is used to generate an optimized flight and sensing schedule for the drones that aims to minimize the total expected execution time.

We built a prototype of BeeCluster and evaluated it on five real-world case studies with drones in outdoor environments, measuring speedups from 11.6% to 23.9%.

CCS CONCEPTS

• **Computer systems organization → Robotics**; **Sensors and actuators**; **High-level language architectures**.

ACM Reference Format:
Songtao He, Favyen Bastani, Arjun Balasingam, Karthik Gopalakrishna, Ziwen Jiang, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. 2020. BeeCluster: Drone Orchestration via Predictive Optimization. In *The 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '20), June 15–19, 2020, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3386901.3388912

1 INTRODUCTION

Rapid progress in the development of small aerial drones has enabled numerous aerial sensing applications, including infrastructure and agricultural inspection [8, 18, 21, 29, 39], air pollution sensing [9, 37, 44, 45], cartography and geographic mapping [26, 33], traffic monitoring [34], disaster relief [7, 35], and search and rescue [16]. However, developing auto-pilot applications that control a fleet of drones to perform complex sensing tasks is often challenging. As a result, most drones today are manually flown by individual pilots or by simple auto-pilot applications [3, 4] that can only handle static tasks, which is impractical for applications that need to react with the environments [20], such as localizing the source of air pollution or searching a target in an unknown environment.

To simplify multi-drone application development and deployment, we propose *BeeCluster*, a drone orchestration system that manages a fleet of drones on behalf of an application. In BeeCluster, application developers write their program for *virtual drones*. The framework then determines how to schedule the drones to minimize the application execution time. At the heart of BeeCluster is *predictive optimization*, in which the run-time framework builds a model to forecast future application tasks, and uses these forecasts to optimize its route planning.

Prior Work: Prior drone orchestration platforms [27, 28, 32, 42, 50] typically adopt a *location-oriented* programming model, where developers provide a set of locations and associated actions, e.g., take photos at specific locations. The system then determines an efficient route to visit all these locations with the available drones, and dispatches the drones to execute the application. A key limitation in all these systems is that they only take into account the current set of requests from the application for route planning. Even frameworks that allow applications to issue new requests dynamically [32] are myopic, and plan routes based only on the current requests at any point in time. By contrast, predictive optimization considers both the current requests and the application's likely future requests in the route planning process. In particular, many applications generate new requests once a given sensing task is complete, or cancel a

```
1 A,B,C = initial locations
2 while True:
3   values = SenseAtLocations({A,B,C})
4   A,B,C = Update(A,B,C,values)
```

Figure 1: The pseudo code of an active sensing loop.

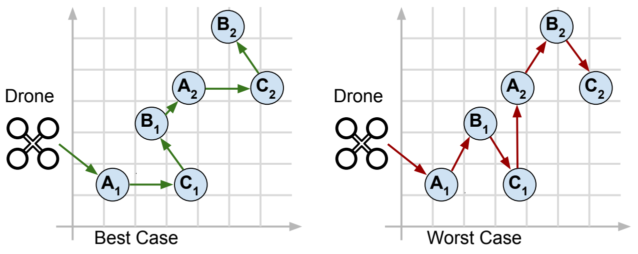


Figure 2: Possible routes in the best case and the worst case using existing drone orchestration systems

request based on the results of a current task. BeeCluster accounts for these likely future actions in the planning process.

The Opportunity: As an illustrative example, consider an application with the active sensing loop shown in Figure 1. The algorithm starts with three initial locations A, B, C. In line 3, the algorithm collects sensor readings from these three locations. Then it updates the three locations based on the sensor readings and proceeds to a new iteration. This basic algorithm represents a category of iterative, active sensing applications, e.g., using gradient descent to localize the source of air pollution [51]. Let A_n, B_n, C_n denote the locations of A, B, C at line 3 in the n-th iteration.

Consider the scenario in Figure 2 where we wish to run this application using one drone. When the program first reaches line 3, the orchestration system dispatches the drone to visit locations A_1, B_1, C_1. At this time, any orchestration system that only considers the current set of requests, i.e., A_1, B_1, C_1, yields two equivalent routes: $A_1 \longrightarrow B_1 \longrightarrow C_1$ or $A_1 \longrightarrow C_1 \longrightarrow B_1$. However, these two routes are not equivalent if we consider multiple iterations of the program. If we choose the route $A_1 \longrightarrow B_1 \longrightarrow C_1$, then in the second iteration, the drone needs to fly from C_1 to A_2 to start the new iteration (the worst case in Figure 2). However, if we choose the route $A_1 \longrightarrow C_1 \longrightarrow B_1$, then in the second iteration, the drone only needs to fly from B_1 to A_2 to start the new iteration (the best case in Figure 2). In this example, the worst case can take 50% more flying time compared to the best case. However, this optimization is possible only if we consider future requests.

We find there are many drone applications that could benefit from predictive optimization. We provide more examples in Section 2.

Challenges: Making drone orchestration systems predictive is not trivial. On the surface, predicting future application requests appears to require an accurate understanding of application semantics and goals. A naive solution to circumvent this challenge is to provide primitives for application developers to explicitly declare possible future requests, or take over the route planning entirely. However, this shifts the burden to application developers, and largely eliminates the advantages of the drone orchestration systems. Thus, we seek a predictive optimization method that does not require developer assistance.

Our Approach and Contributions: Our approach is inspired by the branch prediction in CPUs [40], where the CPU predicts the branches and prefetches potential next instructions to speed-up execution. These predictive optimizations happen without the application changes. The predictive optimization technique we develop for drone optimization works similarly.

BeeCluster has two main components, the API and the runtime. The API provides a virtual-drone programming interface to express a wide range of complex drone applications in a compact and flexible way.

The runtime interprets the application's logic as a dynamic task graph (DTG) and stores the DTG of each execution. While running, BeeCluster uses the historical DTGs as well as the current DTG to forecast the future behaviour of the application (task creations and cancellations). Then, BeeCluster uses this predicted information to minimize the expected execution time.

We have implemented BeeCluster and describe five case studies built atop it, including road mapping, Wi-Fi mapping and hotspot localization, and continuous object tracking. We found that the BeeCluster API is conveniennt to use and that predictive optimization speeds-up execution time by between 11.6% and 23.9% in these case studies.

2 MOTIVATING EXAMPLES

In this section, we show five examples to motivate the benefit of predictive optimizations and the API proposed in BeeCluster. Each example represents a common application pattern.

2.1 Benefits of Predictive Optimization

Example 1: Speculative Execution. Consider the simple active sensing loop shown in Figure 3. In each iteration, the algorithm senses data at location A. Then, the algorithm uses the data to compute the next location to sense data.

Now suppose we have two drones for this application. Current systems will use only one drone on this application at any given time because only one request is outstanding.

```
A = initial location
while True:
  value = SenseAtLocation(A)
  A = Update(A, value)
```

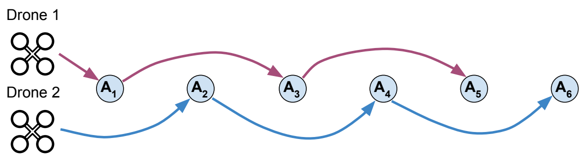


Figure 3: Example of an active sensing loop that can be optimized by speculative execution

By contrast, BeeCluster forecasts the future requests of an application. When there are spare drones in the system, BeeCluster dispatches drones to the locations of the predicted requests, a form of *speculative execution.* Figure 3 shows the traces of the speculative execution on this application with two drones. This strategy overlaps the flying time of one drone with sensing time of another and thus reduces the total execution time.

We find that the code structure in Figure 3 is common in many drone applications, such as localizing the source of air pollution via gradients [51] and mapping trails or roads using iterative tracing [11]. BeeCluster is able to apply this optimization without requiring any changes to application code. We performed an evaluation with a road mapping application that maps a newly constructed road by iteratively tracing the road, and found that this strategy reduces the execution time by 21.3% (case study 1 in Section 5.1.1).

Example 2: Dynamic Request Cancellation. Besides the dynamic request generation in the active sensing examples, the opposite behaviour, *dynamic request cancellation*, is also common in many applications. We show an example in Figure 4. The algorithm needs to measure air pollution at two locations. However, once one location is visited, the application may cancel the sensing request at the other location depending on the first value.

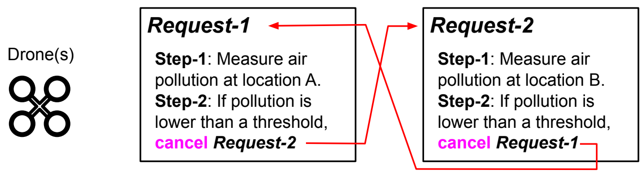

Drone(s)
Request-1
Step-1: Measure air pollution at location A.
Step-2: If pollution is lower than a threshold, cancel Request-2
Request-2
Step-1: Measure air pollution at location B.
Step-2: If pollution is lower than a threshold, cancel Request-1

Figure 4: Example of dynamic request cancellation.

The optimal strategy in this example depends on the prediction of the *if* branch at Step-2. BeeCluster forecasts the potential cancellation for each active request in the system. Then the scheduler can use this forecast information to optimize the route. Examples of applications that can benefit from this optimization include Gaussian-process-based active information gathering [38, 46, 47] for magnetic field, air pollution sensing, wireless signal strength measurement, and exploration in unknown environments [14].

We evaluated an application that maps the Wi-Fi signal strength through Gaussian process (using request cancellations), showing a 23.8% execution time reduction with BeeCluster (case study 2 in Section 5.1.2).

Example 3: Efficient Routing. As noted in the introduction and shown in Figures 1 and 2, knowledge of the future task graph allows BeeCluster to compute the optimal path that minimizes execution time. We evaluated an application that localizes a Wi-Fi hot-spot through gradient descent, finding that BeeCluster's predictive optimization approach improves performance by 11.6% (case study 3 in Section 5.1.3).

2.2 Benefits of the BeeCluster API

Example 4: Fine-Grained Multiplexing. We find that programming APIs in existing drone orchestration systems often bind a virtual drone to a physical drone at task level. The BeeCluster API provides a way to define the binding relationship between virtual drones and physical drones in a precise and fine-grained way, enabling *action-level scheduling*. For example, consider a delivery task that requires a drone to fly from A to B. In *action-level scheduling*, the drone assigned to this task can also perform other tasks, e.g., taking a photo, along the way from A to B. BeeCluster's API allows developers to describe the logic in this example through a fine-grained binding relationship definition (see Section 3.2.2), enabling fine-granularity action-level multiplexing.

We performed an evaluation with a proof-of-concept scenario which involves three simple applications, showing the fine-grained multiplexing can improve the performance by 19.1% (case study 4 in Section 5.2.1).

Example 5: Continuous Operation. Consider the problem of continuously tracking one or more moving vehicles in a city. Because BeeCluster can predict the locations where an algorithm needs to visit in the future, it can dispatch another drone in advance to the location where the original drone may run out of its battery. Then, the original drone hands off the operation to the new drone right before it runs out of battery. Although the hand-off may incur a short delay, it is likely that an operation such as object tracking can still proceed normally. We demonstrate the effectiveness of this feature in Section 5.2.2 (case study 5). Although existing drone orchestration systems promise to provide functional virtualization over many physical drones, the time scale of a single continuous operation is still limited by the battery-time of a single drone.

3 DESIGN

Figure 5 provides an overview of the BeeCluster architecture. The design of BeeCluster is driven by two goals: first, the system should be flexible enough to accurately express a wide variety of drone application logic, and second, the system should be able to effectively optimize applications transparently, i.e., without requiring code changes.

To achieve the first goal, we propose the BeeCluster API (Section 3.1). The second goal is challenging for two reasons. First, many optimization solutions require application-specific knowledge that cannot be obtained easily. Second, the optimization is often application-specific, making it challenging to develop a unified approach to optimize different applications.

To overcome these challenges, BeeCluster uses two ideas. First, it models the application's future behavior as a dynamic task graph (DTG) constructed from the running application. The BeeCluster API makes it possible to do this without application developers needing to codify future behavior. The DTG captures all the necessary information about the application. BeeCluster stores both the DTGs from past runs and the partially constructed DTG of the current run as profiling data for the application. Then, BeeCluster forecasts the application's future behaviour by matching the current DTG with previous DTGs. We describe the details of the DTG and the forecast method in Section 3.2.

Second, rather than providing a single optimization model for all applications, BeeCluster adopts the *instantaneous-assignment* [25] scheduling model and provides an extensible platform to support different optimization heuristics as plugins. Here, the optimization heuristics are used to determine the best *instantaneous-assignment* of tasks to drones at any point in time. For example, one optimization heuristic could prefer to always dispatch drones to their nearest tasks, and in multi-drone scenarios, one optimization heuristic could prefer to scatter the drones over the region of interest. Thus, BeeCluster is not a fixed collection of optimization heuristics. Instead, it provides a common interface that makes it easy to

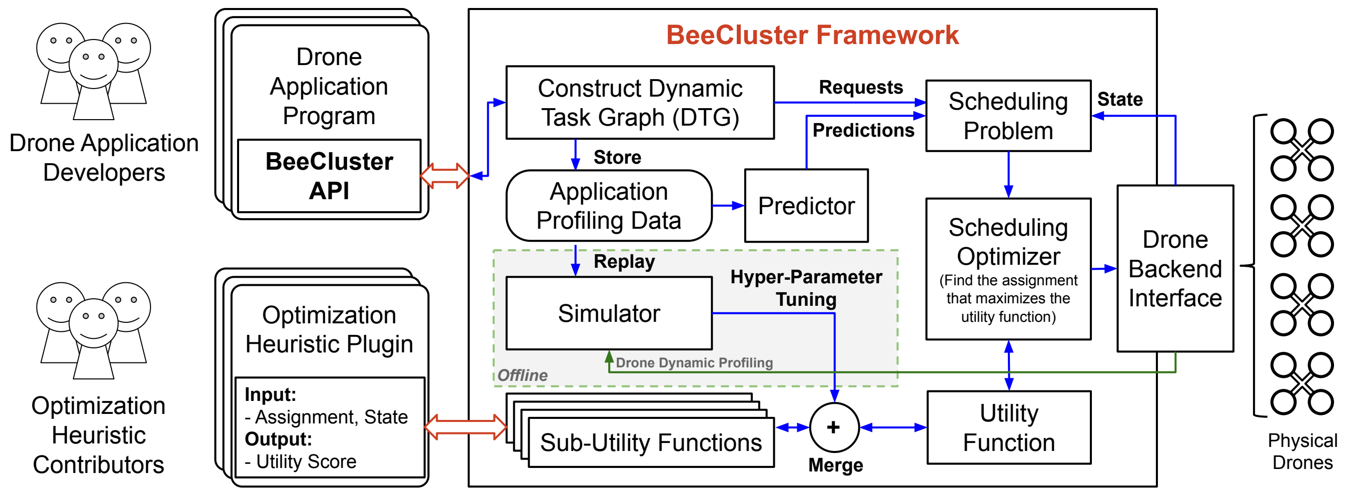


Figure 5: Overview of BeeCluster Drone Orchestration Framework

add new optimization heuristics as plugins. At runtime, BeeCluster considers all possible optimization heuristics and selects the best weighted combination of different heuristics via off-line application replay in a simulator (Section 3.3).

3.1 Programming Model (BeeCluster API)

BeeCluster API allows developers to describe their application logic through a DTG-based imperative programming model. It also allows developers to explicitly define the precise binding relationship between virtual drones and physical drones, e.g., some actions must be done on one physical drone sequentially, while some other actions can be done with multiple physical drones in parallel.

3.1.1 Basic Primitives. We show the basic primitives of the BeeCluster API in Table 1. The API has two basic building blocks, *actions* and *tasks*.

Action. An *action* is a basic drone operation such as taking a photo or flying to a location. The non-blocking property of the action primitive described in Table 1 enables action batching, allowing the scheduler to consider a whole batch of the actions together, rather than processing them one-by-one.

Task. A *task* contains an ordered sequence of actions and maintains the context of a virtual drone; when there are two consecutive actions such that action-1 is flying to location A, and action-2 is taking a photo, the BeeCluster runtime will update the location of the virtual drone after action 1 and interpret the second action as *taking a photo at location A.*

Developers use *newTask()* to create new tasks. Task creation is non-blocking; the blocking (synchronization) only happens when the execution result of a task is required by another statement, i.e., a statement that depends on the execution result of the task. In BeeCluster, each task corresponds to an independent thread, and the thread can be executed in parallel with other threads when there are available drones. The task primitive allows developers to describe independent action sequences, e.g., the application needs to take photos at a set of locations, but the order of execution does not matter. In this case, traveling to a location and taking a photo would be a single task, and there would be one task for each location in the set.

Developers can use *cancelTask()* to cancel tasks specified by the *task_handlers*.

3.1.2 Binding Relationships. The developers can define the binding relationship of the actions within a task. The binding relationship provide a way to control the mapping of virtual drones to the physical drones. BeeCluster supports four binding relationships defined by the combinations of two flags.

SameDrone Flag (SmDrn). When the *SameDrone* flag is set to be True, all the actions within the task need to be done on one physical drone. Otherwise, the actions within the task can be mapped to different physical drones.

Interruptible Flag (Intrp). When the *Interruptible* flag is set to be False , all the actions within the task need to be done one after another, without a large gap in time (best-effort). Otherwise, there could be large gap in time, e.g., 5 minutes, between two consecutive actions.

We show a code example in Figure 6 where there are three tasks with different binding relationships. We show a possible drone schedule in Figure 7.

Task 1. Task 1 has two actions - taking photos at location A and C. The actions can be executed on different drones (*SmDrn=False*) and the time gap between two actions is not restricted (*Intrp=True*).

Task 2. Task 2 is a simple package delivery example. It requires the drone to pick up a package at location A and then drop it at location B. These two actions (pick up and drop) need to be executed on the same drone (*SmDrn=True*). However, the task can be interrupted (*Intrp=True*). After the package is picked up at location A, the drone can be scheduled to perform other actions before flying to location B.

Task 3. Task 3 is a continuous object tracking example where the drone is programmed to track a moving object. Each tracking step can be executed on different drones (*SmDrn=False*) but the time gap between two tracking steps should be minimized by the

Name	Description
h=**act**(*args*)	Execute the action defined by *args* on a drone. Return an action handler *h*. This function call is **non-blocking** but **in-order** - an action is executed when all its predeceased actions in the current thread have been completed.
h=**newTask**(*flags, func, args*)	Create a new task (**a new thread**) with the entry point as function *func* with arguments *args*. The *flags* parameter defines the *binding relationship* (See Section 3.1.2) of the task. This function is **non-blocking**; it returns a task handler *h* immediately after the function call.
result=*h*.**value**	Retrieve the result from a handler *h*. This operation is **blocking**.
cancelTasks(*task_handlers*)	Cancel the tasks specified by the *task_handlers*.

Table 1: BeeCluster API (Minimal Set)

```
def task1(): # Take two photos
  act("flyto", loc_A)
  p1 = act("take_photo")
  act("flyto", loc_C)
  p2 = act("take_photo")
  return p1.value, p2.value # blocking
def task2(): # Package Delivery
  act("flyto", loc_A)
  act("pick_up_package")
  act("flyto", loc_B)
  done = act("drop_package")
  return done.value # blocking
def task3(): # Continues object tracking
  loc = initial_loc
  while True:
    act("flyto", loc)
    photo = act("take_photo").value
    loc = track_and_update(photo, loc)
t1=newTask(task1,SmDrn=False,Intrp=True)
t2=newTask(task2,SmDrn=True,Intrp=True)
t3=newTask(task3,SmDrn=False,Intrp=False)
```

Figure 6: Example of different binding relationships.

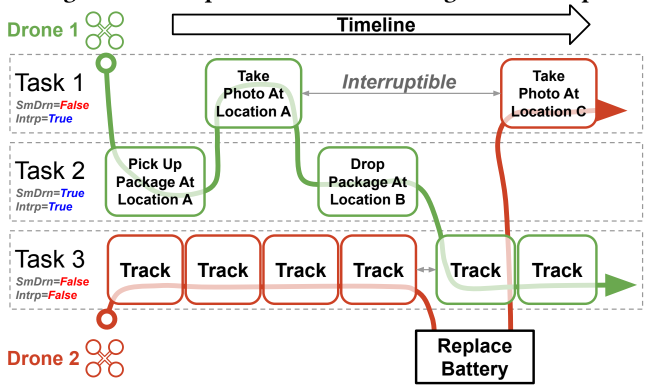


Figure 7: Fine-granularity multiplexing in BeeCluster

system to maintain high tracking success rate; thereby, the *Intrp* flag is set to *False*.

Through defining the binding relationship of tasks, developers can describe the resource requirements of their application precisely. This ability to specify binding relationships makes inter-task multiplexing possible at a fine granularity – for example, one drone can carry out subtasks of several tasks when *Intrp=True*. This improves utilization of the drones, as shown in case study 4 in Section 5.2.1.

3.2 Application Forecasting

BeeCluster forecasts the application's behavior through using the dynamic task graph (DTG) of the running application and by matching the current DTG with previous historical DTGs (i.e., application profiling data).

We show the source code of an example drone application in Figure 8 as well as its DTG. The application iteratively tracks the source of air pollution using a gradient descent algorithm. At each iteration, if the air pollution is greater than a threshold, the application takes one photo at any one of the four locations involved in gradient computing step.

In this section, we use this example to describe the details of **(1)** the structure of DTG, **(2)** and how we use the DTG to forecast the future behaviors of an application.

3.2.1 Dynamic Task Graph (DTG). A dynamic task graph contains nodes and edges. Edges (directional) in a dynamic task graph represent the dependency relationship among nodes. Nodes have two types, task-nodes and synchronization-nodes.

A task-node corresponds to an instance of the task building block in BeeCluster API. For example, in Figure 8, on line 14 (iteration 1), the application creates four tasks based on the function defined at line 3 with different input arguments. These four tasks correspond to the first four task-nodes in the dynamic task graph on the right side of Figure 8. Each task node contains the meta information for the task, including the first location to visit (if it exists), as well as the the primitives (*act, newTask, cancelTasks*) within the task and its duration.

A synchronization-node is used to represent the synchronization behavior of an application. Synchronization nodes are introduced by the use of the *h.value* API call. For example, on line 15, the execution is blocked until all the four tasks created in line 14 are completed. In the dynamic task graph, this synchronization behavior is represented as a synchronization node, i.e., the blue rectangle node in Figure 8.

The DTG doesn't directly represent branches, e.g., the branch at line 16. When there is a branch, the dynamic task graph only contains the path that the application takes. For example, in the first two iterations, the sensing results don't meet the branch condition at line 16. In this case, the DTG doesn't contain the execution path

```
 1 handles = [None,None,None,None]
 2 def task1(loc):     # Line 2-4 "Task1" in the DTG →
 3   act("flyto", loc)
 4   return act("measure_pollution").value
 5 def task2(loc, n): # Line 5-10 "Task2" in the DTG →
 6   wait_until_non_none(handles)
 7   act("flyto", loc)
 8   val = act("action2").value
 9   cancelTasks([handles without handles[n]])
10   return val
11 loc = initial_loc, rng = [0,1,2,3]
12 for i in range(10):
13   locs = four_corners(loc, "10 meters")
14   tasks = [newTask(task1,locs[j]) for j in rng]
15   measurements = [tasks[j].value for j in rng]]
16   if avg(measurements) > threshold:
17     handles = [newTask(task2,locs[j],j) for j in rng]
18     photos = [handles[j].value for j in rng]]
19     handles = [None,None,None,None]
20     StorePhoto(photos)
21   loc = GradientDescent(locs, measurements)
```

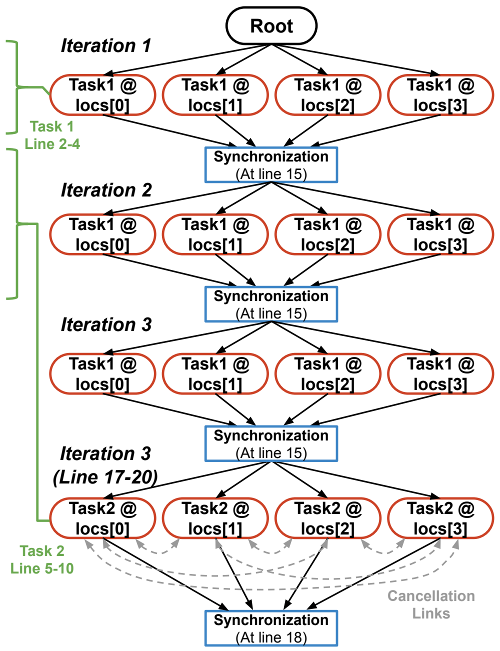


Figure 8: Dynamic Task Graph (DTG) Example.

of line 17-20 for the first two iterations because they don't get executed.

BeeCluster's runtime constructs the DTG of an running application on the fly. The construction algorithm supports concurrent BeeCluster API calls from different application threads, allowing the developers to describe complicated application dependency logic.

3.2.2 *Forecasting using DTGs.*

BeeCluster forecasts the future behaviour of a running application through matching the most recent portion (sub-graph) of the current active DTG with historical DTGs, including **(1)** the current active DTG without its frontier nodes (leaf-nodes) , **(2)** and the DTGs from past runs. Once a match is found, we can simply use what happened next in the historical DTGs as a forecast for what may happen next in the current application run.

Graph-Match and Forecast Algorithm. We call our forecasting algorithm GMF, for **G**raph **M**atch and **F**orecast. The GMF algorithm contains two phases.

In the first phase, GMF searches for accurate sub-graph matches through an incremental matching procedure. GMF starts with a sub-graph containing only one node (a frontier node) in the current DTG. This sub-graph is used as the target sub-graph for matching. Then, GMP finds all the one-node sub-graphs in the historical DTGs that match with this target sub-graph. All the matched one-node sub-graphs are considered as candidate sub-graphs. Here, we say two task-nodes matched when they are from the same logic task but could have different input arguments.

After this, GMF starts to iteratively increase the depth of both the target sub-graph and the candidate sub-graphs. At each iteration, GMF removes unmatched sub-graphs in the candidate sub-graphs from previous iteration. As the depth increases, the number of the candidate sub-graphs decreases. This incremental matching procedure stops when the number of the candidate sub-graphs is below a threshold (10 in our implementation), or when the depth of the sub-graph exceeds a threshold (10 in our implementation).

In the second phase, GMF looks into the meta information of each task-node and ranks all the matched candidate sub-graphs through a similarity metric. One meta information we used here is the first location a task visited, e.g, for a task that took a photo at location A, the first location it visited is location A.

Inspired by [48, 49], we use a rotation-and-translation-invariant distance, between the first locations of all the task-nodes in two matched sub-graphs as the similarity metric; given two sub-graphs, we apply a spatial transformation that only contains rotation and transition to one of the sub-graph so that the total distance between the first locations of each node pairs is minimized. This minimized distance is the rotation-and-translation-invariant distance between two sub-graphs. It is easy to extend this similarity metric to also consider the sensor readings, e.g., using cosine-similarity for image embeddings for image data captured at each task-node.

Finally, GMF outputs what happened next from the top-K, i.e., top-10, matched sub-graphs as a forecast. Here, we apply the spatial transformation to the predicted task-nodes so that they have the correct locations.

3.3 Extensible Optimization Heuristics

In this subsection, we first describe the BeeCluster's scheduling model (Section 3.3.1), then we show how BeeCluster supports different optimizations as plugins in Section 3.3.2.

3.3.1 *Scheduling Model.*

BeeCluster's scheduling model uses *instantaneous assignment* [25] of tasks to drones; in this model, the scheduler only needs to decide the next request each drone needs to handle. The scheduler takes the current states of the drones, the current visible requests, and the forecast requests as input, and outputs an instantaneous assignment (a set of drone-to-request pairs).

At high-level, the scheduler is triggered to reschedule drones when its input is changed. In our implementation, we carefully choose when to reschedule to avoid system overhead; in particular, we try to batch changes to the input to eliminate frequent rescheduling.

Inside the scheduler, the scheduler searches for the best instantaneous assignment that maximizes the utility function. Here, the utility function is used to evaluate the goodness of the instantaneous assignment. Similar to the strategy used in Google's routing library [6] and the drone delivery problem [17], the scheduler first computes a valid assignment using a greedy algorithm. Then, the scheduler starts a local search from the greedy assignment; it keeps applying random permutations (e.g., swap the assigned tasks of two drones) to the current assignment and update the current assignment toward a lower-cost assignment in each iteration. Often, a meta-heuristic is needed in this local search phase to help escape from local minima. Here, we use simulated annealing [41].

In BeeCluster, this utility function is a linear combination of different sub-utility functions. Although the global goal of BeeCluster is to minimize the execution time of the application, we achieve that through optimizing the instantaneous assignment at any point in time.

Each sub-utility function implements one optimization heuristic (we list a few below) that evaluate the goodness from one aspect. These sub-utility functions take the current configuration (both drones and requests) and an instantaneous assignment as input and outputs the utility (goodness) of the input instantaneous assignment. Here, we list a few sub-utility functions (optimization heuristics) as examples.

Closest Next Request Heuristic. This is the simplest greedy heuristic. The sub-utility function outputs the negative total estimated flying time of all drone-to-request pairs in the instantaneous assignment.

Avoid Mutual Utility Heuristic. When there are two requests A and B in an application, and if A is done, B will be cancelled, if B is done, A will be cancelled. In this case, the system should not dispatch two drones to A and B at the same time. We implemented an optimization heuristic similar to the utility function in [14] to avoid having this happen. This heuristic relies on the forecast information provided by BeeCluster.

Avoid Zigzag Heuristic. We use this heuristic to avoid the worst case in Figures 1,2. This optimization heuristic also relies on the forecast information provided by BeeCluster. The sub-utility function computes the geometric center of all forecast and visible requests. Then, the sub-utility function causes the drones to visit the farthest request from this center by assigning a higher utility score to each drone-to-request pair when the request is far away from the geometry center.

Speculative Execution Heuristic. When there are spare, idle drones in the system, this heuristic encourages the spare drones to fly to their closest forecast requests. This heuristic is similar to the *Closest Next Request Heuristic*, but it operates only on forecast requests.

Fairness Heuristic. This sub-utility function outputs the negative total waiting time of the involved requests in the drone-to-request pairs of the instantaneous assignment. This heuristic intends to give high priority to the requests that have long waiting time.

3.3.2 *Optimization Heuristic Plugins and Automatic Plugin Balancing.*

There exists many other optimization heuristics. As an extensible platform, BeeCluster provides an interface for optimization heuristic contributors so that they can easily add new optimization heuristics as plugins into the BeeCluster platform.

When there are multiple optimization heuristics available in a system, it is hard to figure out what combination (weights) of these heuristics can make a specific application run faster. In BeeCluster, we make the hyper-parameter tuning automatic. After several initial runs of an application, BeeCluster replays the executions of the application using the historical DTGs in a simulator and finds the linear weights for the different sub-utility functions (optimization heuristics) that minimizes execution time using hyper-parameter search [19].

In BeeCluster, we use *hyperopt* library [12] to search the hyper-parameters. Hyper-parameter search techniques have been extensively studied in AutoML research. Using hyper-parameter search in our context is practical because there are fewer hyper-parameters compared with the hyper-parameters in the AutoML problem and the sampling cost (i.e., DTG replay) is much less than the sampling cost (i.e., training a new model) in the AutoML problem.

4 IMPLEMENTATION

In this section, we show the implementation details of BeeCluster. We discuss the software in Section 4.1 and Section 4.2, and the hardware in Section 4.3.

4.1 BeeCluster Framework

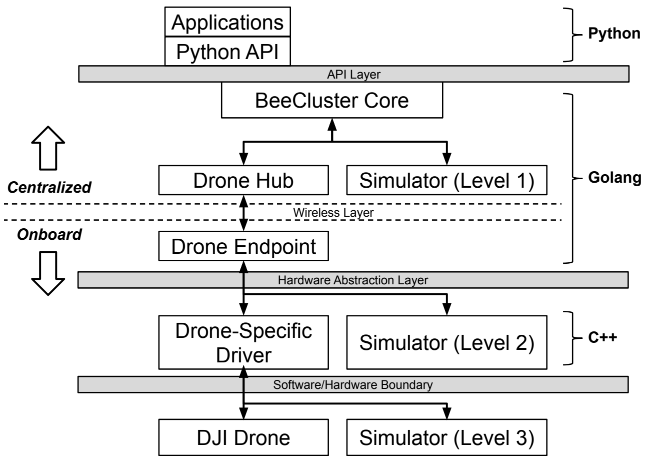


Figure 9: BeeCluster Framework

BeeCluster is released as open-source (http://beecluster.csail.mit.edu). The implementation of the BeeCluster framework comprises ≈ 15K lines of code (LoC); 80.8% in Golang for the BeeCluster core system, 5.3% in python for the python wrapper of BeeCluster API, and 13.9% in C++ for the drone driver. We show a diagram of the BeeCluster component architecture in Figure 9.

Simulators. The implementation of BeeCluster involves three simulators. The level-1 simulator is used to conduct fast application replay for hyper-parameter search. The level-2 simulator is used to verify the correctness of the system and the application before take-off. The level-3 simulator is DJI's hardware simulator, which is used to verify the correctness of the drone driver code. These simulators boost the development speed of BeeCluster and avoid potential failures in real-world experiments.

Drone-to-centralized controller communication protocol. In the implementation, the drone sends a heart-beat message to the centralized controller every 100 ms. The centralized controller encloses the action descriptions (if any) into the return message. Once the drone receives the return message, it performs the actions enclosed in the return message and sends the result back to the centralized controller together with the next heart-beat message.

Collision avoidance and conflicts resolution (CACR). We implement a simple centralized first-come-first-serve CACR protocol in BeeCluster to avoid drone collisions and resolve conflicts. This CACR protocol is critical for real-world drone deployments.

4.2 BeeCluster Task

BeeCluster tasks run as threads. Each task instance has two threads, one is the main thread executing the task code, the other is a helper thread. Each task maintains its task context. The task context consists of an action queue and the status of the virtual drone (e.g., the location of the drone, and the id of the bound physical drone if existed).

When the code inside a task thread calls the **act()** function, the runtime system pushes the action into the action queue of the task. The helper thread of the task keeps polling the action queue and execute the action through communicating with the BeeCluster server.

Failure Handling. An action may fail due to the *change-of-state* of drones. For example, while a physical drone is executing an action, its battery level drops below a critical threshold. The low battery level than triggers the return-to-home protection mechanism and marks the state of the drone as "unavailable" to the system. In this case, the action fails.

BeeCluster handles action failures through retrying the failed actions or calling the user-defined failure handlers. The failure handling behavior depends on the drone binding flags of the task. When the *SmDrm* flag and *Intrp* flag are not set to *SmDrm=True, Intrp=False* at the same time. The helper thread retries the failed action until success.

When *SmDrm=True, Intrp=False*, retrying the failed action may change the semantic of the task. For example, suppose we have a video recording task consisting of four actions, (1) fly to location A, (2) start recording, (3) fly to location B, and (4) stop recording. If the third action fails, we can not simply retry it alone, instead, we have to retry the whole sequence of actions. Handling failure in this case often involves user-defined failure handling logic. So when *SmDrm=True, Intrp=False*, BeeCluster forwards the failure to a user-defined failure handler. In the failure handler, users can decide if they want to retry the whole task or not.

BeeCluster also supports a special user-defined failure type - *timeout*. When a task times out, BeeCluster cancels the task and calls the user-defined error handler, regardless of the binding flags of the task.

Override Binding Relationship Flags. In BeeCluster API, the drone binding flags are associated with each logical task rather than each function. We made this design choice because we want to reuse the function code, e.g., a function could be assigned with different drone binding flags dynamically. However, we find this design choice could be error-prone. For example, developers may forget to set the binding flags when they create a new task that can only run with some specific flags. To address this issue, we introduce the *setFlags()* API. If a function doesn't need to be reused, developers can use *setFlags()* inside the function to override the passed-in binding flags of a task.

4.3 Hardware Setup

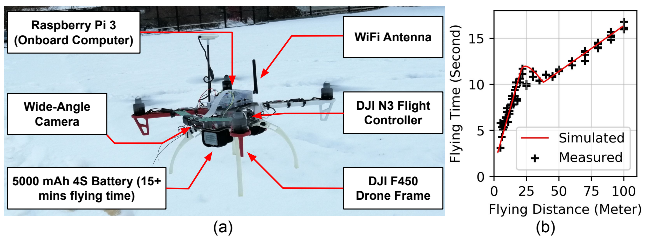


Figure 10: Drone Hardware Used in Evaluation

Similar to other drone orchestration system, BeeCluster is drone-agnostic. In our prototype, we use a custom assembled drone platform as our drone back-end for evaluation. We show our drone platform in Figure 10(a). Our drone platform is based on the DJI F450 drone frame [1]. We use DJI N3 flight controller [2] with a GPS antenna. The flight controller is connected to an on-board Raspberry Pi single-board computer [5]. Each drone is equipped with a 2dBm WiFi antenna, which enables communication (>150 meters) between the drone and the centralized controller. Each drone is also equipped with a wide angle camera for camera-based applications.

Drone profile. To make the simulator accurate, we profile the dynamic (acceleration rate, de-acceleration rate and drag coefficient) of our drones (Figure 10(b)). In our setup, the drone flies at 2 m/s when the target is within 20 meters, otherwise, the drone flies at 10 m/s.

5 EVALUATION

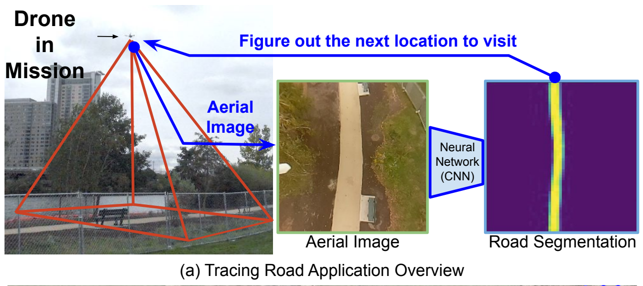
In our evaluation, we seek to answer the following questions:

- What is the benefit of BeeCluster's application-agnostic predictive optimization?
- What is the benefit of BeeCluster's API?
- What is the overhead of BeeCluster?

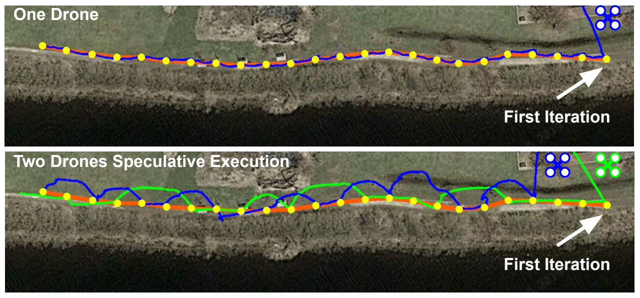
To address these questions, we evaluate BeeCluster with five case studies, consisting of four representative applications and one proof-of-concept scenario consisting of three simple applications. We conduct most of our experiments with real drones and real environments. In addition, we use our level-2 simulator to conduct simulated experiments that complement our real-world experiments.

In the evaluation, we create a baseline solution that shares the same API as BeeCluster but uses a simple greedy routing algorithm. We use this simple baseline solution to represent other drone orchestration systems as most of the exiting solutions either use simple greedy routing algorithms or don't support dynamic tasking.

5.1 Benefit from predictive Optimizations



(a) Tracing Road Application Overview

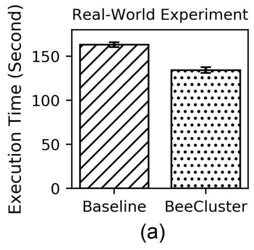


(b) Real-World Experiments with One and Two Drones

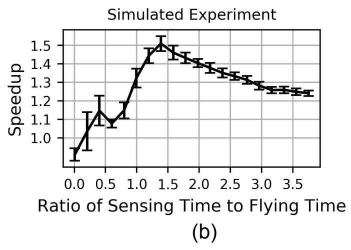
Figure 11: Case Study 1: Mapping a Newly Constructed Road through Iterative Tracing

5.1.1 *Case Study 1: Road Mapping.* Automatically mapping new constructed roads is an important real-world task [10, 11, 22, 30, 31], e.g., for companies that maintain digital maps. In case study 1, we built a drone-based mapping application with the BeeCluster API where the drones can smartly map a newly constructed road through iterative road tracing. We show this application in Figure 11(a). The application begins at the start of the road. In each iteration, the application acquires a photo at the current location and computes the road position and direction through a neural network. Then, the application moves 6 meters along the road and starts the next iteration.

We evaluate this application on a newly constructed trail in a nearby park. We run the application for 25 iterations, which covers 144 meters of the trail. We show the drone trajectories in Figure 11(b). We highlight the positions of the trail at each iteration using a yellow circle.



(a)



(b)

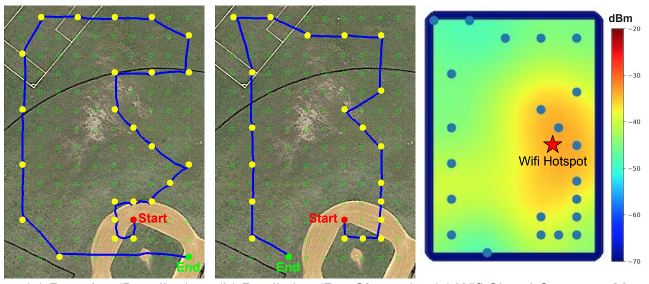
Figure 12: Benefits from Speculative Execution.

Benefit: The application logic in this case study is an active sensing loop (see Section 2.1.1) with strong sequential dependencies; the i-th iteration depends on the execution result of the $(i-1)$-st iteration.

While the application is running, BeeCluster forecasts the locations where the application may need to visit in the future. When there are spare drones in the system, BeeCluster dispatches the spare drones to the forecast locations. This speculative execution strategy overlaps the sensing time and flying time of the active sensing loop, reducing the total execution time of the application. To show this, we added a second drone. We show drone trajectories in Figure 11(b). In the two-drone scenario, we can clearly see that the two drones alternatively fly over each other. Here, the curvy traces are the result of our collision avoidance and conflict resolution protocol.

We repeat the experiments three times for both the one-drone scenario and two-drone scenario. As shown in Figure 12(a), BeeCluster's speculative execution strategy improves the sensing task runtime by 21.3%. Here, the baseline approach does not support speculative execution, instead, it only executes the program line by line. In fact, the benefit of the speculative execution heavily depends on the ratio of sensing time to flying time. We simulated different sensing time to flying time ratios for the tracing road application in a simulator. As shown in Figure 12(b), we find speculative execution performs the best when the ratio of sensing time to flying time is close to 1.5, which achieves up to 50% performance improvement.

In this case study, we don't use DTGs from previous runs; the forecast is only based on the early portion of the active DTG, e.g., the first few iterations. We find this setup is sufficient because the application is predictable and the speculative execution does not require very accurate predictions to overlap the flying time and sensing time.



(a) Reactive (Baseline) Flight TIme: 270 seconds (b) Predictive (**BeeCluster**) Flight TIme: 228 seconds (c) Wifi Signal Coverage Map (Gaussian Process)

Figure 13: Case Study 2: Mapping Wifi Signal Coverage through a Gaussian Process

5.1.2 *Case Study 2: Wifi Coverage Map.* In case study 2, we built an application with the BeeCluster API to create a WiFi coverage map of an open area. Our application adapts an efficient and popular information gathering algorithm based on Gaussian Processes [36, 38].The algorithm first divides the region of interest into an equal sized grid (i.e., 5 meters by 5 meters). Then, the algorithm issues tasks (using the *newTask* primitive) to request a WiFi signal strength measurement from all grid cells. When the WiFi signal strength is measured at a grid, the algorithm updates the Gaussian Process model (which models the WiFi signal strength field) and updates the uncertainty estimate on all other grids. If the uncertainty estimation of a grid falls below a threshold, the measurement request in that grid is cancelled (using the *cancelTask* primitive). The algorithm stops when all the grid cells are measured or have satisfied the uncertainty threshold.

We evaluate this application in a 50 meter by 70 meter open area. In Figure 13(a,b), we show the center of each 5 meter by 5 meter grid cell in green and highlight the visited grid cells in yellow. We show the estimated WiFi coverage map in Figure 13(c).

Benefit: In this case study, BeeCluster forecasts the task cancellation behavior of the application. Here, we use one additional historical DTG from a previous run in the simulator. We find the best weights of different optimization heuristics through replaying this DTG in a simulator. We find this simulated DTG can be transferred to our real-world environment. Then, inside the scheduler, the *Avoid Mutual Utility Heuristic* (Section 3.4.1) uses the forecast information to encourage drones to visit the grids that maximize the expected information gain. We can clearly see the effect of this predictive optimization heuristic in Figure 13(a,b). In the baseline (only with reactive optimization), the drone visits grids on the boundary of the region. In contrast, BeeCluster's predictive optimization strategy causes drones to visit grids that are at least one grid away from the boundary. This is because once a grid is visited, its neighboring grids will be cancelled with a high probability. Note that BeeCluster captures this insight without requiring explicit input from the application, simply using the Avoid Mutual Utility heuristic.

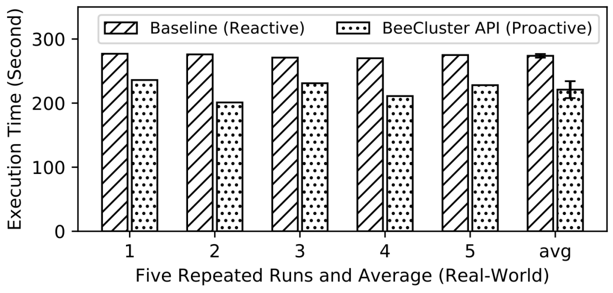


Figure 14: BeeCluster enables *Application Agnostic predictive Optimization*, reducing the execution time by 23.9% on average over five runs in case study 2

We repeat the experiments 5 times with real drones and real environments. As shown in Figure 14, BeeCluster's predictive optimization yields an 23.9% average runtime improvement in this scenario.

5.1.3 *Case Study 3: WiFi Hotspot Localization.* For case study 3, we built an application to locate the WiFi hotspot using gradient descent. The algorithm is similar to the code shown in Figure 8 - without the *if* statement from lines 16 to 20.

We evaluate this application with real drones and real environments. In the evaluation, we set the initial location of the algorithm to be 60 meters away from the hotspot and then ran the algorithm for ten iterations. In each iteration, the current location moves 5 meters toward the direction of the gradient. We show an example trajectory in Figure 15(a).

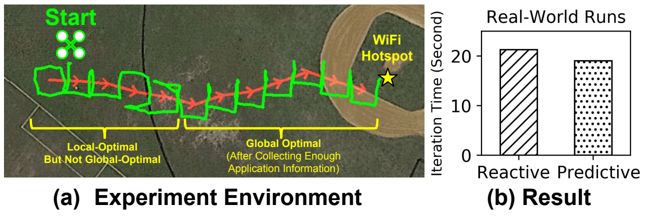


Figure 15: Case Study 3: WiFi Hotspot Localization

Benefit: In this case study, BeeCluster can forecast the new tasks generated by the applications. As the four measurements in each iteration don't have an order, the baseline solution (reactive) may not always pick up the best order (clockwise or counter-clockwise) in which to perform the measurements. In contrast, BeeCluster leverages forecasting information to always pick the best order. We repeat the experiments 5 times. We show the average execution time of the last 5 iterations in Figure 15(b). BeeCluster's predictive optimization improves the performance by 11.6% against reactive optimization.

In this case study, we don't use DTGs from previous run; the forecast is only based on the early portion of the active DTG, e.g., the first few iterations. As shown in Figure 15(a), the drone didn't always follow the global optimal route in the first few iterations. Later on, once the system collected enough application profiling data (DTG), the drone started to always follow the global optimal route.

5.2 Benefit from BeeCluster API

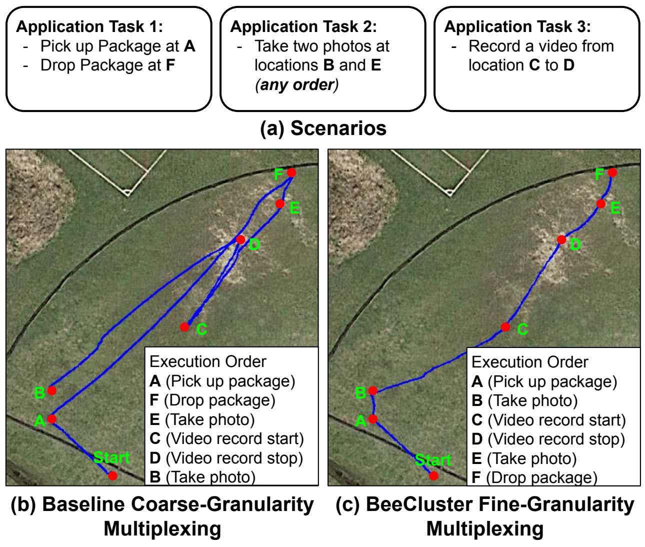


Figure 16: Case Study 4 Fine Granularity Multiplexing

5.2.1 *Case Study 4: Fine Granularity Multiplexing.* In this case study, we built a proof-of-concept scenario to demonstrate the benefit of BeeCluster's precise binding relationships. We create three fake application tasks (Figure 16(a)) with different binding relationships. The first application task mimics the package delivery task. It needs to pick up a package at A and drop it at F. The task requires the same drone to do the pickup and dropoff but the task is interruptible. The second application task takes two photos at two locations B and E. The two photos can be taken in any order (implemented with *newTask* primitive). The third application task record a video from location C to D. The task requires the same drone to record and the task is not interruptible.

In the evaluation, we submit these three application tasks to the drone orchestration system (with one drone). We fix the locations A and F and randomly generate the other four locations to create 10 different random scenarios. In the baseline orchestration system, we disable the precise binding definition in application task 1 but keep the application tasks 2 and 3 the same. In this case, once the application task 1 starts, the drone is occupied until the application task 1 is completed.

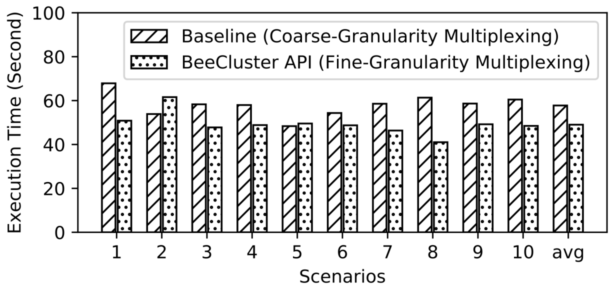


Figure 17: BeeCluster API enables *fine granularity multiplexing*, reducing the execution time by 19.1% on average over ten runs in case study 4

Benefit: We repeat the experiment 10 times with real drones. We show the quantitative result in Figure 17. BeeCluster surpasses the baseline by 19.1% in terms of execution time. This improvement comes from fine-granularity multiplexing; as shown in Figure 16(c), BeeCluster can multiplex application task 1 with other tasks; once the drone picks up the package at location A, it doesn't need to send the package to location F immediately, instead, the drone can still conduct other tasks along the way to location F.

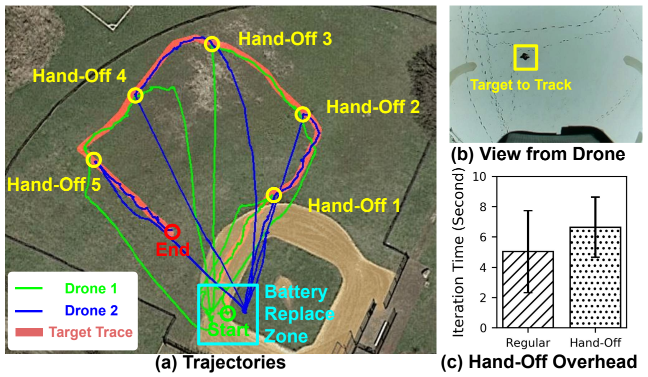


Figure 18: Case Study 5: Continuous object tracking beyond a single drone's battery life.

5.2.2 *Case Study 5: Continuous Object Tracking.*

For case study 5, we built an application to continuously track a person. The duration of the tacking task is longer than the battery-time of a single drone. We use this case study to demonstrate the effect of the *non-same drone* but *uninterruptible* binding relationship (see Section 3.2.2). The application logic is similar to line 13-18 in Figure 7.

We use two real drones in this experiment. We artificially limit the flying time of each drone to be only 60 seconds. After the battery is used up, the drone needs to fly back home to simulate recharging its battery (in reality the drones have about 30 minutes of battery life, so actual battery replacement wasn't necessary). In the experiment, we track a moving person for 5 minutes (with 5 drone hand-offs). We show the trajectories of the two drones in Figure 18(b).

Benefit: We demonstrate that BeeCluster can support continuous operation beyond a single drone's battery time without any additional code - but with a small overhead; we show the average execution times of each regular iteration and each hand-off iteration in Figure 18(c). Here, the hand-off iterations are the iterations where the hand-off happened.

5.3 BeeCluster Overhead

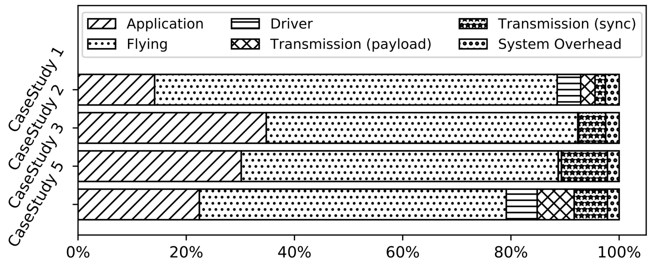


Figure 19: Execution Time Breakdown

Next, to study the overhead of the BeeCluster framework, we measured the execution time of each component for case studies 1,2,3, and 5. Case study 4 is a proof-of-concept scenario, so we don't consider it in this measurement. In this measurement, we use only one drone and repeat each application three times. We logged the timestamps at the entries and exits of each components on both the drone side and the central controller side. Then, we merged all the logs together to create the execution time breakdown of each components. We show this execution time breakdown for the four applications in Figure 19.

There are six parts in the execution time breakdown. *Application time* is the time spent within the application (python), including the `sleep` function call. *Flying time* is the amount of time when the drone is flying toward target. *Driver time* is the amount of time spent inside the driver, e.g., taking a photo may take 100-200 ms. *Transmission time* consists of two parts: *transmission (payload) time* is the time spent sending the data between the drone and the centralized controller; *transmission (sync) time* is the delay introduced by the fixed rate (synchronized) drone-to-centralized controller communication protocol (See Section 4.1). On average, this protocol introduces a 150 ms (100 ms + 50 ms) delay for each action.

The overhead of BeeCluster, which includes dynamic task graph construction, application forecasting, and scheduling, is the difference between the five components described above and the total wall-clock execution time[1]. The average system overhead of the four applications is 2.34%, which we believe is acceptable.

Scalability For completeness, we show the scheduling time and the forecast time with different numbers of drones and historical DTGs in Figure 20. Here, the results are from case study 2 for scheduling time, and from case study 3 for forecast time because these two applications have the highest overhead in scheduling and forecast, respectively. We use level-2 simulator to measure these overhead.

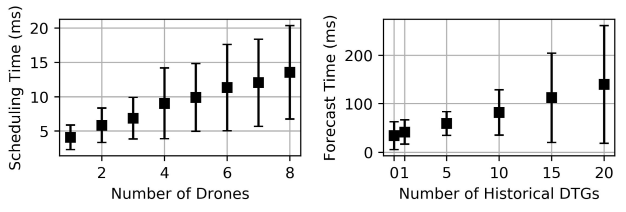


Figure 20: Scheduling Time and Forecasting Time

6 RELATED WORK

Work related to drone orchestration has been done in different fields, including computer systems, robotics, and operations research. Figure 21 illustrates how BeeCluster is different from existing work.

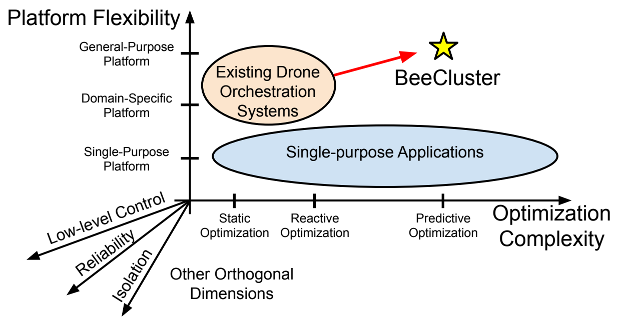


Figure 21: Positioning of BeeCluster vs Related Work

In contrast to prior drone orchestration systems such as AnDrone [42], Voltron [32], and UAV-as-a-service [27, 28, 50], our main contribution is developing a *predictive* optimization strategy, in contrast the non-predictive nature of prior work. In addition, we designed a novel general-purpose programming API that allows developers to describe the logic of a wide range of drone applications in a precise way. This new API enables our predictive optimizations.

In contrast to single-purpose drone applications [9, 17, 24, 37, 38, 43], BeeCluster provides predictive optimization in a transparent way. Unlike many single-purpose applications or algorithms, which, if they are predictive, require application developers to provide the information needed to make predictions, BeeCluster infers the necessary information from the program execution and profiling data.

[1]This is because we only use one drone.

There are other important aspects in a drone orchestration system that have been studied, including low-level drone control [13], reliable communication protocols [15, 23], and isolation [42]. This work is complementary and orthogonal to BeeCluster.

7 DISCUSSION

Dynamic Task Graph. In the core of BeeCluster, we choose the dynamic task graph (DTG) as BeeCluster's programming model because this model can capture a wide range of applications' logic and provide high programming flexibility. For example, BeeCluster API can be used in a wide range of coding structures and models such as the conditional branch, loops, recursive functions, graph traversal (graph search), multi-threading models, and etc.

Moreover, in the design of BeeCluster, DTG acts as the intermediate representation (IR) that bridges the frontend and backend. Using a general-propose IR can benefit future works, e.g., advancements in the backend can benefit all applications without the need of changing the application code.

Application Forecasting. BeeCluster forecasts the application behavior in an automatic way to reduce the programming complexity. However, the accuracy of the forecast could be a concern. Although we can improve the forecast accuracy by improving the backend of BeeCluster, an alternative design choice does exist. We can let users express the future behavior of the application explicitly (e.g., using code annotation). In another word, we can trade programming complexity for forecast accuracy. We think studying the property of this trade-off is desired in future work.

Low-level Primitives. BeeCluster uses a set of low-level primitives. For example, the maneuver of a drone (e.g., fly to a location) and the action of a drone (e.g., take a photo) are split into two primitives. We made this design choice because we want the primitives to be indivisible, thereby reduce the complexity of the backend.

One concern about this design choice is that the backend may not be able to use the full information of a high-level action. For example, if a high-level action requires the drone to visit a sequence of locations in order, the backend may only be able to see one location at a time, therefore lose the opportunity of using the whole sequence of locations to do optimization. However, BeeCluster overcomes this issue thanks to the non-blocking design of the action API (see section 3.1).

8 CONCLUSION

In this work, we described a new drone orchestration system, *BeeCluster*. BeeCluster is able to take into account the future sensing tasks that applications will execute when making scheduling decisions. This is achieved through a novel programming API that allows developers to describe the logic of a wide range of drone applications in a precise and compact way, as well as an extensible rule-based forecasting approach. We demonstrate the effectiveness BeeCluster through a evaluation over five real-world case studies with real drones in outdoor environments, demonstrating speedups ranging from 11.6% to 23.9%.

ACKNOWLEDGMENTS

We thank the anonymous reviewers whose comments helped improve this manuscript and complemented the design of BeeCluster.

REFERENCES

[1] Dji f450 drone frame. https://www.dji.com/flame-wheel-arf.

[2] Dji n3 flight controller. https://www.dji.com/n3.

[3] Dronedeploy. https://www.dronedeploy.com.

[4] Pix4dcapture. https://www.pix4d.com/product/pix4dcapture.

[5] Raspberry pi. https://www.raspberrypi.org/.

[6] Vehicle routing problem solvers in google or-tools. https://developers.google.com/optimization/routing/routing_options. Accessed: 2019-09-17.

[7] Adams, S. M., and Friedland, C. J. A survey of unmanned aerial vehicle (uav) usage for imagery collection in disaster research and management. In *9th International Workshop on Remote Sensing for Disaster Response* (2011), vol. 8.

[8] Adão, T., Hruška, J., Pádua, L., Bessa, J., Peres, E., Morais, R., and Sousa, J. Hyperspectral imaging: A review on uav-based sensors, data processing and applications for agriculture and forestry. *Remote Sensing 9*, 11 (2017), 1110.

[9] Alvear, O., Zema, N. R., Natalizio, E., and Calafate, C. T. Using uav-based systems to monitor air pollution in areas with poor accessibility. *Journal of Advanced Transportation 2017* (2017).

[10] Bastani, F., He, S., Abbar, S., Alizadeh, M., Balakrishnan, H., Chawla, S., and Madden, S. Machine-assisted map editing. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2018), ACM, pp. 23–32.

[11] Bastani, F., He, S., Abbar, S., Alizadeh, M., Balakrishnan, H., Chawla, S., Madden, S., and DeWitt, D. Roadtracer: Automatic extraction of road networks from aerial images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), pp. 4720–4728.

[12] Bergstra, J., Yamins, D., and Cox, D. D. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In *Proceedings of the 12th Python in science conference* (2013), Citeseer, pp. 13–20.

[13] Bregu, E., Casamassima, N., Cantoni, D., Mottola, L., and Whitehouse, K. Reactive control of autonomous drones. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services* (2016), MobiSys '16.

[14] Burgard, W., Moors, M., Stachniss, C., and Schneider, F. E. Coordinated multi-robot exploration. *IEEE Transactions on robotics 21*, 3 (2005), 376–386.

[15] Chlestil, C., Leitgeb, E., Schmitt, N. P., Muhammad, S. S., Zettl, K., and Rehm, W. Reliable optical wireless links within uav swarms. In *2006 international conference on transparent optical networks* (2006), vol. 4, IEEE, pp. 39–42.

[16] Doherty, P., and Rudol, P. A uav search and rescue scenario with human body detection and geolocalization. In *Australasian Joint Conference on Artificial Intelligence* (2007), Springer, pp. 1–13.

[17] Dorling, K., Heinrichs, J., Messier, G. G., and Magierowski, S. Vehicle routing problems for drone delivery. *IEEE Transactions on Systems, Man, and Cybernetics: Systems 47*, 1 (2016), 70–85.

[18] Eschmann, C., Kuo, C.-M., Kuo, C.-H., and Boller, C. Unmanned aircraft systems for remote building inspection and monitoring. In *Proceedings of the 6th European Workshop on Structural Health Monitoring, Dresden, Germany* (2012), vol. 36.

[19] Feurer, M., and Hutter, F. *Hyperparameter Optimization*. Springer International Publishing, Cham, 2019, pp. 3–33.

[20] Floreano, D., and Wood, R. J. Science, technology and the future of small autonomous drones. *Nature 521*, 7553 (2015), 460–466.

[21] Ham, Y., Han, K. K., Lin, J. J., and Golparvar-Fard, M. Visual monitoring of civil infrastructure systems via camera-equipped unmanned aerial vehicles (uavs): a review of related works. *Visualization in Engineering 4*, 1 (2016), 1.

[22] He, S., Bastani, F., Abbar, S., Alizadeh, M., Balakrishnan, H., Chawla, S., and Madden, S. RoadRunner: Improving the precision of road network inference from gps trajectories. In *ACM SIGSPATIAL* (2018).

[23] Ho, D.-T., and Shimamoto, S. Highly reliable communication protocol for wsn-uav system employing tdma and pfs scheme. In *2011 IEEE Globecom Workshops (Gc Wkshps)* (2011), IEEE, pp. 1320–1324.

[24] Julian, K. D., and Kochenderfer, M. J. Distributed wildfire surveillance with autonomous aircraft using deep reinforcement learning. *Journal of Guidance, Control, and Dynamics* (2019), 1–11.

[25] Korsah, G. A., Stentz, A., and Dias, M. B. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research 32*, 12 (2013), 1495–1512.

[26] Lin, Y., Hyyppa, J., and Jaakkola, A. Mini-uav-borne lidar for fine-scale mapping. *IEEE Geoscience and Remote Sensing Letters 8*, 3 (2010), 426–430.

[27] Mahmoud, S., Mohamed, N., and Al-Jaroodi, J. Integrating uavs into the cloud using the concept of the web of things. *Journal of Robotics 2015* (2015), 10.

[28] Mahmoud, S. Y. M., and Mohamed, N. Toward a cloud platform for uav resources and services. In *2015 IEEE Fourth Symposium on Network Cloud Computing and Applications (NCCA)* (2015), IEEE, pp. 23–30.

[29] Máthé, K., and Buşoniu, L. Vision and control for uavs: A survey of general methods and of inexpensive platforms for infrastructure inspection. *Sensors 15*, 7 (2015), 14887–14916.

[30] Máttyus, G., Luo, W., and Urtasun, R. Deeproadmapper: Extracting road topology from aerial images. In *Proceedings of the IEEE International Conference on Computer Vision* (2017), pp. 3438–3446.

[31] Máttyus, G., Wang, S., Fidler, S., and Urtasun, R. Hd maps: Fine-grained road segmentation by parsing ground and aerial images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 3611–3619.

[32] Mottola, L., Moretta, M., Whitehouse, K., and Ghezzi, C. Team-level programming of drone sensor networks. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems* (2014), ACM, pp. 177–190.

[33] Nex, F., and Remondino, F. Uav for 3d mapping applications: a review. *Applied geomatics 6*, 1 (2014), 1–15.

[34] Puri, A. A survey of unmanned aerial vehicles (uav) for traffic surveillance. *Department of computer science and engineering, University of South Florida* (2005), 1–29.

[35] Quaritsch, M., Kruggl, K., Wischounig-Strucl, D., Bhattacharya, S., Shah, M., and Rinner, B. Networked uavs as aerial sensor network for disaster management applications. *e & i Elektrotechnik und Informationstechnik 127*, 3 (2010), 56–63.

[36] Rasmussen, C. E. Gaussian processes in machine learning. In *Summer School on Machine Learning* (2003), Springer.

[37] Rossi, M., Brunelli, D., Adami, A., Lorenzelli, L., Menna, F., and Remondino, F. Gas-drone: Portable gas sensing system on uavs for gas leakage localization. In *SENSORS, 2014 IEEE* (2014), IEEE, pp. 1431–1434.

[38] Ruiz, A. V., Angermann, M., Wieser, I., Frassl, M., and Mueller, J. Efficient multi-agent exploration with gaussian processes. In *Australasian Conference on Robotics and Automation (ACRA)* (2014).

[39] Saari, H., Pellikka, I., Pesonen, L., Tuominen, S., Heikkilä, J., Holmlund, C., Mäkynen, J., Ojala, K., and Antila, T. Unmanned aerial vehicle (uav) operated spectral camera system for forest and agriculture applications. In *Remote Sensing for Agriculture, Ecosystems, and Hydrology XIII* (2011), vol. 8174, International Society for Optics and Photonics, p. 81740H.

[40] Smith, J. E. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture* (1981), IEEE Computer Society Press, pp. 135–148.

[41] Van Laarhoven, P. J., and Aarts, E. H. Simulated annealing. In *Simulated annealing: Theory and applications*. Springer, 1987, pp. 7–15.

[42] Van't Hof, A., and Nieh, J. Androne: Virtual drone computing in the cloud. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), ACM, p. 6.

[43] Vasisht, D., Kapetanovic, Z., Won, J.-h., Jin, X., Chandra, R., Kapoor, A., Sinha, S. N., Sudarshan, M., and Stratman, S. Farmbeats: An iot platform for data-driven agriculture. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (2017).

[44] Villa, T., Gonzalez, F., Miljievic, B., Ristovski, Z., and Morawska, L. An overview of small unmanned aerial vehicles for air quality measurements: Present applications and future prospectives. *Sensors 16*, 7 (2016), 1072.

[45] Villa, T., Salimi, F., Morton, K., Morawska, L., and Gonzalez, F. Development and validation of a uav based system for air pollution measurements. *Sensors 16*, 12 (2016), 2202.

[46] Viseras, A., Shutin, D., and Merino, L. Online information gathering using sampling-based planners and gps: an information theoretic approach. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), IEEE, pp. 123–130.

[47] Viseras, A., Wiedemann, T., Manss, C., Magel, L., Mueller, J., Shutin, D., and Merino, L. Decentralized multi-agent exploration with online-learning of gaussian processes. In *2016 IEEE International Conference on Robotics and Automation (ICRA)* (2016), IEEE, pp. 4222–4229.

[48] Vlachos, M., Gunopulos, D., and Das, G. Rotation invariant distance measures for trajectories. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (2004), ACM, pp. 707–712.

[49] Werman, M., and Weinshall, D. Similarity and affine invariant distances between 2d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence 17*, 8 (1995), 810–814.

[50] Yapp, J., Seker, R., and Babiceanu, R. Uav as a service: Enabling on-demand access and on-the-fly re-tasking of multi-tenant uavs using cloud services. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)* (2016), IEEE, pp. 1–8.

[51] Yungaicela-Naula, N. M., Zhang, Y., Garza-Castañon, L. E., and Minchala, L. I. Uav-based air pollutant source localization using gradient and probabilistic methods. In *2018 International Conference on Unmanned Aircraft Systems (ICUAS)* (2018), IEEE, pp. 702–707.