Programming Micro-Aerial Vehicle Swarms With Karma

Karthik Dantu*
Harvard University
Cambridge, MA, USA
kar@eecs.harvard.edu

Bryan Kate*
Harvard University
Cambridge, MA, USA
bkate@eecs.harvard.edu

Jason Waterman
Harvard University
Cambridge, MA, USA
waterman@eecs.harvard.edu

Peter Bailis
UC Berkeley
Berkeley, CA, USA
pbailis@cs.berkeley.edu

Matt Welsh
Google, Inc.
Seattle, WA, USA
mdw@mdw.la

Abstract

Research in micro-aerial vehicle (MAV) construction, control, and high-density power sources is enabling swarms of MAVs as a new class of mobile sensing systems. For efficient operation, such systems must adapt to dynamic environments, cope with uncertainty in sensing and control, and operate with limited resources. We propose a novel system architecture based on a *hive-drone* model that simplifies the functionality of an individual MAV to a sequence of sensing and actuation commands with *no in-field communication*. This decision simplifies the hardware and software complexity of individual MAVs and moves the complexity of coordination entirely to a central hive computer. We present *Karma*, a system for programming and managing MAV swarms. Through simulation and testbed experiments we demonstrate how applications in Karma can run on limited resources, are robust to individual MAV failure, and adapt to changes in the environment.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; I.2.9 [**Artificial Intelligence**]: Robotics—*Autonomous Vehicles*

General Terms

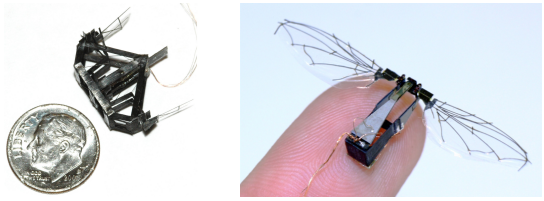
Design, Management

Keywords

Swarm, Micro-Aerial Vehicle, Mobile Sensor Network

* Co-primary author

SenSys'11, November 1–4, 2011, Seattle, WA, USA.



(a) MFI (b) RoboBee

Figure 1. Example prototypes of insect-scale MAVs.

1 Introduction

Micro-aerial vehicles (MAVs) are an emerging class of sensing systems consisting of small autonomous aerial vehicles, capable of limited computing, communication, sensing, and actuation [8, 19]. At the forefront of MAV research are efforts to construct *insect-scale* flapping-wing MAVs, just a few centimeters in diameter. Recent advances in airframe construction [22, 24], flight dynamics and control [20], sensor design [4, 13], and high-density power sources [14] are quickly pushing insect-scale MAVs closer to mass production. Figure 1 depicts two such prototypes.[1]

A swarm of such MAVs could be considered an airborne sensor network, an approach that opens up many novel research directions in terms of applications and systems infrastructure to support the swarm. Applications include urban surveillance, crowd monitoring, and disaster recovery, where they can be quickly deployed and monitored from a nearby ground station [16]. Insect-scale MAVs are most appropriate for situations requiring covert sensing, operation in confined spaces, or fine-grained manipulation of the environment.

Mobility allows a MAV swarm to cover a much larger area than a stationary sensor network and to reposition the coverage as the features of interest change in the environment. However, MAV swarms also present a unique set of challenges. Actuation dominates the weight and power budgets for these devices, keeping sensing and control to the bare minimum. The limits on sensing and control imply that MAVs will often fail in the field. This can be mitigated by

[1] Images courtesy of R. J. Wood (MFI) and Ben Finio (RoboBee)

the sheer size of the swarm, but as the swarm size scales up it becomes harder to reason about the swarm as a whole and the complexity of coordination increases.

The extreme resource limitations of the insect-scale MAV platform restrict the complexity of the programs that can be executed on the vehicle. In this work, we focus on constructing high-level programs from simple MAV behaviors. More specifically, we present a system for controlling the actions of MAV swarms that is built around our investigation of the following questions:

- How can the behaviors that execute on MAVs be simplified to match the resource constraints while maintaining global utility to the swarm?
- In what ways can simple MAV behaviors be combined to accomplish a swarm-level goal?
- To what extent can the burden of low-level MAV coordination be lifted from the end user?

We propose a system architecture based on the *hive-drone model* in which individual MAVs, called *drones*, perform the simple sensing and manipulation tasks required to fulfill the goals of the swarm. We impose the restriction that drones *can not* communicate with each other in the field. Further, we assume that drones operate *without precise knowledge of location*, relying on proprioception or periodic external localization. These restrictions simplify drone programs and allows swarm-scale behavior that is coordinated by a centralized *hive*, which contains sufficient sensing, computation, and storage capabilities to manage the swarm. With the majority of the computational burden pushed to the hive, the application programmer can focus on implementing the correct behaviors for the application (what to do) while the system reasons about the execution of the application on the swarm (where and when to do it).

To achieve this decoupling, we propose a programming model that allows the application programmer to compose programs from simple MAV-level behaviors by relating the behaviors that produce information to the ones that consume it. This model allows for an easy and flexible composition of programs and enables the system to reason about MAV scheduling efficiently.

Our contribution in this paper is Karma, a system that demonstrates the feasibilty of a centralized approach to programming and managing large-scale MAV swarms. We show the capabilities of our system in simulation and on prototype hardware. Our system is efficient with respect to time and energy, resilient to failure, and adaptable to changing environmental conditions.

2 Background: MAV Swarms

Several research groups are developing insect-scale MAV platforms [8, 9, 24], some of which are shown in Figure 1. MAVs in this class are typically only a few centimeters in diameter, with the total weight limited by the amount of thrust that can be generated by the wings. Studies of insect-scale MAV prototypes have shown that up to 93% of the weight budget (50 mg total) and 95% of the power budget (10 mW total) is allocated for actuation [11]. These restrictive budgets impose hard constraints on the computing and sensing elements. In addition to sensing for control, the MAV must carry sensors required to perform tasks for the intended application. For example, in a commercial crop pollination scenario, the MAV requires a sensor to detect flowers [13] and an appendage to collect and deposit pollen.

Insect-scale MAV platforms have two advantages – extremely small size and deployment in large numbers. Insect-scale MAVs are relatively inconspicuous and can operate in enclosed, close-quartered areas where traditional aerial vehicles cannot fly. In addition, these systems can be used to perform tasks that are challenging for larger platforms, such as landing on a flower and collecting or depositing pollen. Further, hundreds to thousands of MAVs can be deployed in conjunction to achieve a specific task in a massively parallel fashion. A MAV swarm makes up for lack of sophisticated sensing and actuation through scale in deployment, providing parallelism and robustness to failure in the field.

In this work we target applications that closely match the strengths of MAV swarms. Applications that are well suited to the swarm approach exhibit a set of common characteristics:

- **Spatial Concurrency**: The application requires a certain *coverage* of the target area (e.g. surveillance or environmental monitoring). Swarms take a massively parallel approach to achieve this coverage.
- **Approximate Operation**: Due to sensing and actuation limitations, applications often take a stochastic approach to navigation and task execution. It may not be possible for a single MAV to reliably sense a given location, but many MAVs sensing in a target region are likely to collect the desired information in aggregate.
- **Mobility**: MAV swarm applications exploit near-constant actuation for rapid deployment and dynamic reconfiguration.

2.1 Example Application: Alfalfa Crop Monitoring and Pollination

Throughout this paper we use an example application to illustrate the design and operation of our system. The application, alfalfa crop monitoring and pollination [7], represents a typical application of MAV swarms in that it relies on both information gathering and micro-manipulation behaviors in a relatively static environment. Alfalfa is an important food crop for cattle and requires an external pollinator (e.g. bees) to produce seeds. In recent years, colony collapse disorder [21] has devastated honeybee populations and jeopardized the cultivation of important crops. We believe that a swarm of insect-scale MAVs is well-suited to performing this type of pollination.

In addition to pollination, alfalfa crops require periodic monitoring for pests and disease. These tasks need to be performed at least three times a week, and are normally done with visual spot checks. We envision a full service application that not only pollinates the crop when it is in bloom, but monitors the crop throughout the growing season.

To meet these requirements, the application consists of three periodic behaviors: searching for pests, searching for

diseases, and looking for flowers in bloom. Pest infestation is typically detected by inspecting the leaves of the crop for damage caused by feeding insects. Diseases can be detected by looking at the color of the leaves, which turn greenish-white or brown in the presence of disease. If pests or diseases are found, the system notifies the farmer so that he can treat the infected area. If the flowers are in bloom, the system should start a one-time pollination behavior. Together, the application has four behaviors in total. We will use this application to describe our design and evaluate our system.

3 Hive-Drone Model

In Karma, we seek a system design and programming abstraction that shifts most of the complexity away from the application programmer and towards the underlying system. We propose the *hive-drone model*, which moves the coordination complexity to a centralized computer. In this model, MAVs are stationed at the *hive*, which has a physical presence in the environment and the capability to recharge MAV batteries. The hive computer determines how MAVs should be used to accomplish the swarm objectives and dispatches MAVs as *drones* to execute specific tasks in the environment. This system design is coupled with a programming model that allows the application programmer to specify the desired swarm behaviors as sequences of sensing, sensor processing, and actuation commands without concern for coordination.

In our system, the complexity of coordination and fault tolerance becomes the responsibility of the system, not the application programmer. We minimize the program complexity of individual MAVs by eliminating in-field communication and restricting programs to a simple set of commands. Therefore, by design, the application programmer is granted the freedom to describe how a MAV behaves when it is dispatched, but has no ability to explicitly coordinate its actions in the field. While these restrictions prohibit the expression of some programs, they allow us to accomplish our original goal of presenting a simple swarm interface to the user. Our approach is analogous to the philosophy of the MapReduce programming model [6]; we provide a simple, powerful abstraction to the user with the caveat that not all computations can be expressed using the abstraction. However, if the invariants of the model are followed, the system will handle the inherent complexity of coordinating the massively parallel operation.

Further, the hive-drone model fits well with the target hardware. In addition to the processing and sensing limitations, there is a rather severe restriction on flight time. Current estimates of insect-scale MAV flight times suggest that a drone could operate for 5–10 minutes before its energy source is depleted. If the drone must return to the hive to recharge after a short period of time, it makes sense to eschew in-field communication in favor of implicit coordination at the hive. There are few scenarios that would necessitate in-field communication given the flight time restrictions and inherent burden of complexity.

The benefits of the hive-drone model can be summarized as follows:

- **Simplify MAV programming**: The drones need not coordinate amongst themselves, make tasking decisions, or deal with MAV loss. Thus, their software complexity is reduced.
- **Better decision making:** The centralized hive is more informed than an individual drone making greedy decisions based on partial information. The overhead for sharing information in the field is eliminated. The hive has the advantage of collective intelligence and more computation to better allocate resources.

In any sensing paradigm there will be a delay from the time an event of interest occurs in the environment to when the end user is notified of the event. We call this delay the *information latency* in the system. In traditional sensor network deployments, this latency may be on the order of a few seconds, depending on the duty cycle of the sensor and the properties of the network that propagates the information to a base station. One drawback of the hive-drone model is that it introduces an additional latency because drones retain information collected in the field until they return to the hive rather than routing it through a network. It is up to the user to decide if this latency (on the order of minutes) is tolerable given the nature of their application. There are some optimizations that can be made in the system to reduce this latency at runtime (Section 4.3).

To facilitate the description of our system and the discussion of the hive-drone model, we introduce the following definitions that will be used throughout the paper.

- **Sortie**: One round trip in which a drone executes a single behavior.
- **Behavior**: A sequence of sensing, sensor processing, and actuation commands that are followed by a drone on a sortie. For example, a disease detection behavior for the alfalfa application consists of performing a random walk to search for alfalfa leaves, acquiring an image with a simple image sensor, and checking for diseases with a color matching algorithm.
- **Application**: A composition of low level drone behaviors and high level goals that is submitted by a user for execution on the swarm.

3.1 Spatial Decomposition

Given the inherent spatial nature of swarm applications, it is necessary to control the dispersion of the swarm throughout a target space. At a high level, the user deploying the swarm application has some notion of how it should be distributed. For example, a farmer may desire uniform coverage of the pest monitoring behavior, but want targeted execution of the pollination behavior. The Karma system provides an abstraction that establishes a shared spatial context between the hive and the application behaviors. At the time of deployment, the target area in which the swarm will operate is divided into *regions*. The spatial decomposition may be influenced by application parameters (e.g. the size of the field and the desired sampling resolution) but is ultimately controlled by the hive. Behaviors are assumed to be written in a location agnostic manner so that they can be applied to any

region in the target space. However, they are given access to a location service at runtime that provides the current region for accounting purposes.

The choice to decompose space into regions benefits our system in multiple ways. First, it transforms the area of deployment from a continuous space into a discrete space, making it easier to reason about MAV allocation. Second, it aligns the localization primitives available to the drone behaviors with the likely capabilities of the MAV platform. Given the extreme limitations on computation and sensing, it is unlikely that the MAVs will have access to high-resolution location services in the field. The MAVs will rely on a combination of proprioception and exteroception (e.g. odometry using inertial sensors and a polarized light compass [15]) to navigate in the field. It may also be possible to externally localize the MAVs from the hive using RF triangulation or harmonic radar [18]. This information could be used to update the MAVs in flight or correlate sensor readings with a location when the drones return. In this paper we assume that it is possible to localize MAVs to the resolution of a region through a combination of these techniques, though we do not solve this problem directly.

Finally, it is not necessary for the regions in the spatial abstraction to be defined using a Cartesian coordinate system. The abstraction will work just as well if the regions are defined topologically or as nodes in a graph. For example, it may be possible to embed beacons into a target area and allow the drones to localize to the region that is defined by the closest beacon signal. Though it is an exciting prospect, we do not explore the use of non-Cartesian decompositions in this work.

3.2 Data Model

The hive-drone model advocates for central storage of information that is collected in the field. For this purpose, the hive maintains a key-value repository called the *Datastore*. Updates to this data structure are asynchronous, occurring when drones return from a sortie. The value for each key is structured as a log, with new data appended along with metadata describing the time and location at which it was collected by the drone. Thus, the Datastore can be queried both temporally and spatially (at the resolution of a region). The information collected in the Datastore is used by the hive to track the progress of the application and make allocation decisions.

Figure 2 shows the data model in action. A drone flies out with a blank local store (called a *scratchpad*) that is populated as it executes a behavior. Upon its return, the information it collected is uploaded to the hive Datastore. Thus, the Datastore at the hive has a partial view of the environment at any given time, which is dependent on the information brought back by drones that have completed their sorties.

3.3 Programming Model

As defined above, an application in our system is a composition of low-level drone behaviors and high-level goals. In general, we restrict behaviors to be location agnostic so that they can be applied to any region as determined by the hive. Though we do not restrict the actions taken by the behaviors, we only explore simple algorithms in this work (e.g. random walks, open loop patterns, and periodic sensing).

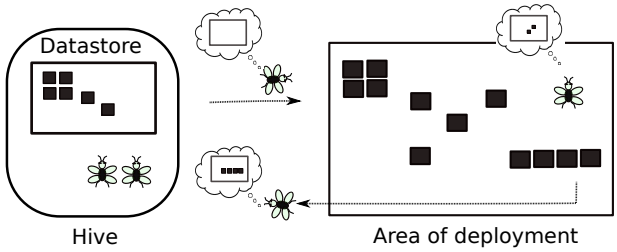


Figure 2. The hive-drone data model in action. Drones are dispatched with blank scratchpads. As they execute a behavior, they populate their scratchpad. On return to the hive, the scratchpad is appended to the Datastore.

The high-level goals of an application are more abstract. In general, the user will need to specify the sequencing of behaviors and the area over which the swarm will operate. For example, a farmer may wish to specify a portion of crops to be monitored and that a pollination behavior should only be executed following the detection of flowers in bloom.

From the perspective of the hive, the sole purpose of a drone behavior is to populate the Datastore with new information. We take this notion one step further and require that every behavior produce some type of information under normal execution. For monitoring behaviors this requirement is trivial to fulfill – a new piece of information can be produced with each sensor reading. A similar approach is taken by behaviors that manipulate the environment. For instance, a pollination behavior could record attempted landings. All behaviors produce information that is propagated to the Datastore and the hive uses this information to reason about the state of the application at runtime. This is the idea behind the Karma programming model, in which this information is used to define relationships between behaviors and selectively apply them to regions. In this abstraction, the programmer defines an application as a set of simple drone behaviors along with two functions for each behavior:

- **Activation Predicate**: A boolean function based on the information in the Datastore. The hive can allocate a drone to execute this behavior if the function evaluates to true.
- **Progress Function**: A function based on the information in the Datastore that evaluates to a real number between 0 and 1, indicating the progress made toward the application goal associated with the behavior. When this function returns a value of 1, the application has achieved the behavior's goal.

These function definitions enable the hive to make decisions about MAV allocation over time and space. Using the activation predicates, the hive can determine *when* it is appropriate to execute each behavior. Since the Datastore can be queried spatially (at a regional resolution), the context of the information passed to this function can be narrowed to determine *where* the behavior is activated. This allows a programmer to specify a data dependency that defines the selective execution of a behavior at runtime. Further, the in-

formation used by the activation predicate defines the prerequisites for execution. That is, an implicit dependency is created between a behavior that produces the information used by another behavior's activation predicate. By defining the activation predicates this way, a programmer can sequence the execution of behaviors. Since each activation predicate is evaluated independently, our model allows multiple behaviors to be activated concurrently, including behaviors that have a dependent relationship. For example, consider our alfalfa application; the pollination behavior is dependent on the bloom monitoring behavior, but both could be activated once some flowers have been found. Pollination could begin on the known flowers while more are sought. We would like to note that the implicit behavior graph created by this representation may contain cycles. At this point, we make no effort to detect cycles in an application.

The progress function is used by the hive to reason about resource allocation. By tracking the rate of change of this function, it can determine an estimate for the number of drones that are required to complete the behavior. Like the activation predicate, the Datastore query can be narrowed to a regional context, allowing the hive to make more targeted drone allocations. By defining the progress function this way, we assume that the application is associating a fixed amount of work with a behavior. Most of the target applications that we have studied require a fixed amount of work or can be transformed into a periodic representation, removing the problem caused by open-ended goals.

3.4 Scheduling Problem

The hive-drone model and our programming abstraction transform the problem of executing an application on a MAV swarm into a problem of scheduling behaviors on drones. There are many policies for determining the allocation of drones to behaviors. We have chosen the *shortest time to application completion* as one objective. There is an economic argument for wanting to finish an application as soon as possible – swarm maintenance (e.g. powering the hive computer and charging drones) may be expensive. Other reasons might be environmental in nature; for example, MAV flight will not be possible when it is raining, so an application may be racing against an unfavorable weather forecast. This objective advocates greedily scheduling all available drones. It could, however, lead to a policy that would execute behaviors that are concurrently activated in a batch sequence. That is, the scheduling objective makes no distinction between a schedule that allocates all drones to behavior `A` until it is completed followed by all drones to behavior `B`, and one that interleaves allocations for `A` and `B`. We posit that concurrently activated behaviors should be scheduled fairly (without starvation) in the absence of prioritization information. A second objective is introduced to *achieve fairness between behaviors*. Fairness can be defined as parity in the output of the progress functions of individual behaviors. Therefore, the second objective of our scheduling function is to minimize the difference in progress between any two activated behaviors.

More formally, let $B = \{b_i \mid 1 \leq i \leq n\}$ be the set of behaviors in the application. Let d be the total number of drones available in the swarm. Let $prev_i$ be the total number of drones that have previously executed behavior b_i. At the current time, let $prog_i(S)$ be the progress made toward the goal of behavior b_i and S the state of the Datastore. Let the estimate of the rate of progress made for behavior i per drone be defined as $rate_i = \frac{prog_i(S)}{prev_i}$. Let $curr_i$ be the number of drones currently running this behavior. Let d be the number of drones currently available for dispatch.

To solve our objectives, we need to allocate $alloc_i$ drones to behavior i such that

$$max\{rate_k * (prev_k + curr_k + alloc_k)\} \; \forall \; k \in [1,n]$$

and

$$min \; \{[rate_i * (prev_i + curr_i + alloc_i)] - [rate_j * (prev_j + curr_j + alloc_j)]\} \; \forall \; (i,j) \; \in \; [1,n]$$

such that

$$c + \sum_{i \in (1,n)} \{alloc_i + curr_i\} \leq d$$

Given these objectives, the hive will allocate drones to behaviors as resources become available.

4 Karma Implementation

We have built Karma, a resource management system for MAV swarms based on the hive-drone model. Figure 3 depicts the functional block diagram of Karma. The Karma runtime at the hive consists of a hive Controller, Scheduler, Dispatcher, and Datastore. The hive Controller is the overall manager of the runtime and invokes the other modules when needed. When a user submits an application to the Karma system, the hive Controller determines the set of active processes (using the activation predicates), and invokes the Scheduler to allocate the available drones to them. The hive Controller monitors the progress of each process and considers the application complete when the set of active processes is empty. The Scheduler is periodically invoked by the Controller to allocate drones to each active process. The Dispatcher is responsible for tracking the status of the physical resources (the MAVs). It programs the drones with the allocated behavior prior to a sortie, tracks the size of the swarm, and notifies the Controller when a drone returns to the hive and is ready for re-deployment.

4.1 Programming the Swarm

To accomplish any goal with the MAV swarm, a user must submit an application. Karma applications are collections of *processes* that are executed at the hive. Each process defined by the application has an associated behavior that is executed on the drone. Each process also specifies the information that it yields when its behavior is executed, as well as the information that is required to activate the process. This is accomplished by enumerating the Datastore keys for the information that is used and yielded. The activation predicate and progress function are evaluated by the Scheduler in the context of queries made against the Datastore using these keys. As mentioned in Section 3.2, these queries can be temporally bounded to suit the needs of the application.

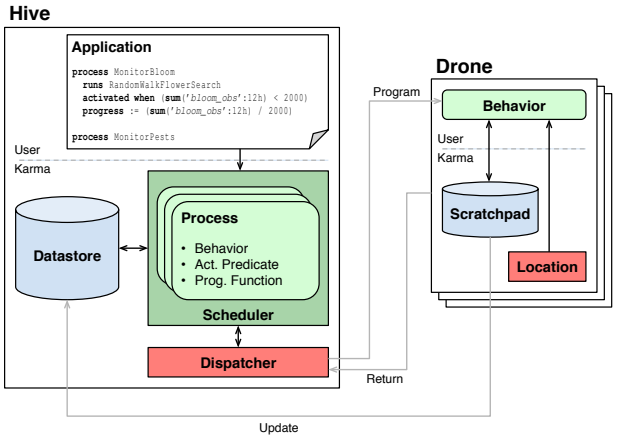


Figure 3. Block diagram of the Karma design. Applications containing sets of processes are submitted to the Karma hive by a user. Each process definition contains an activation predicate, a progress function, and a drone behavior. The Scheduler allocates resources (available drones) to processes. The Dispatcher consumes the allocation and programs behaviors on to drones and dispatches them on a sorties. Upon their return, drones transfer the contents of their scratchpad to the hive Datastore.

```
process MonitorBloom
  runs RandomWalkFlowerSearch
  uses ()
  yields ('bloom_obs', 'bloom_det')
  activated when ('bloom_obs':12h < 2000)
  progress := ('bloom_obs':12h / 2000)

process MonitorPests
  ...

process MonitorDisease
  ...

process Pollinate
  runs RandomWalkPollinate
  uses ('bloom_det')
  yields ('pollinated')
  activated when ('bloom_det':12h > 0)
  progress := (if isNull('bloom_det':12h)
               then 1
               else ('pollinated':12h /
                     (2 * 'bloom_det':12h)))
```

Figure 4. Pseudo-code for the process definitions that make up the alfalfa crop monitoring and pollination application.

Figure 4 shows pseudo-code for part of the example application described in Section 2. The code defines four processes, two of which are omitted for brevity but are defined similarly to `MonitorBloom`. In this case, the pseudo-code references library behaviors that are specified outside of the application code. With this approach it is possible to define common routines that execute on the drone and share them amongst applications. The behavior associated with the `MonitorBloom` process, `RandomWalkFlowerSearch`, executes a random walk pattern while periodically using an optical sensor to detect the presence of flower blooms. Each time the sensor is read, a counter in the drone's local scratchpad with the key *bloom_obs* is incremented. After analyzing the reading, a counter with the key *bloom_det* is incremented if a flower bloom was detected. When the drone returns to the hive, this information is propagated to the hive Datastore, where it can be queried by the Scheduler to evaluate activation predicates and progress functions. In this case, it uses the total number of observations in the past 12 hours to determine if the `MonitorBloom` process should be activated, and how much progress has been made toward its completion. The `Pollinate` process references a similar behavior that executes a random walk and lands the drone on flowers, collecting and depositing pollen through incidental contact. However, unlike the `MonitorBloom` process, which is activated by a lack of observations in the past 12 hours, this process is only activated when flower blooms have been detected (in the past 12 hours). The information dependency on *bloom_det* creates an implicit sequence of operations. For the `Pollinate` process to active, the Datastore must contain information for the key `bloom_det`. In our application, this means that a drone running the `RandomWalkFlowerSearch` behavior for the `MonitorBloom` process must observe flowers on a prior sortie. Alternatively, should the need arise to bootstrap the `Pollinate` process, the user could inject this information into the Datastore manually. The information dependency also exists in the progress function defined by the `Pollinate` process. The function is defined such that the process is considered complete in regions where there is no *bloom_det* information. This allows the progress of the process to be evaluated regionally, and prevents drones from being allocated to regions where no blooms have been detected. Finally, recall that the activation predicate for each process is evaluated independently, potentially resulting in multiple activated processes. In this case, the monitoring processes may execute concurrently with the pollination process, and the Scheduler must allocate resources appropriately.

4.2 Karma Scheduler

Karma implements a fair, work-stealing scheduler that solves the optimization problem illustrated in Section 3.4. It operates in two steps. First, it estimates the total workload to be performed for the set of active processes per region. Let $P = \{p_i \mid 1 \leq i \leq n\}$ be the set of active processes. Let R be the set of regions the area of operation is divided into. Let S_t be the state of the Datastore at the hive at time t. The progress of a process i in a given region r at time t can be evaluated using the progress function provided ($PF_i(S_t^r)$). Let us denote this progress as $prog_{(i,r)}^t = PF(S_t^r)$. At a future time t' ($t' > t$) when the scheduler is invoked, let there be k drones executing sorties for process i in region r that have returned to the hive.

We can compute the rate of progress per drone-sortie for process i ($q_{(i,r)}$) as $q_{(i,r)} = \frac{(prog_{(i,r)}^{t'} - prog_{(i,r)}^{t})}{k}$. Given the rate of progress per drone per sortie $q_{(i,r)}$, we can estimate the amount of work remaining to complete process i in region r

(in terms of number of drone-sorties) as $N^{t'}_{(i,r)} = \frac{(1-prog_{(i,r)})}{E[q_{(i,r)}]}$. Since the value of $q_{(i,r)}$ can vary significantly over time and with environmental conditions, we compute the value $E[q_{(i,r)}]$ using a weighted average of historical $q_{(i,r)}$ values. Note that the progress rates $q_{(i,r)}$ cannot be computed at $t = 0$. We bootstrap the progress estimation mechanism by sending a fixed number of *scout drones* to regions where there is insufficient progress information. Once the progress of the scout drones is known, $q_{(i,r)}$ can be computed as defined above. We can then estimate the total work to be done across all active processes as $N^{t'} = \sum_{i\in(1,n),r\in R} N^{t'}_{(i,r)}$. Note that the work estimate $N^{t'}$ assumes a linear relationship between the number of drones dispatched and the amount of progress made.

The second step is to allocate the available drones to the set of active processes fairly. Karma takes a work-stealing approach to allocation, using a sorted queue with each element in the queue representing an active process requesting drones in a region. The queue is sorted in ascending order according to the *service level* of each request. We define service level as the ratio of remaining amount of work to complete process i in region r ($N^{t'}_{(i,r)}$) to the total work to be done by the application for all active processes across all regions at the current time ($N^{t'}$). As drones become available, Karma allocates them iteratively by servicing the request at the head of the queue. If m drones are available for allocation, this results in an allocation of $alloc^{t'}_{(i,r)}$ for each process i in region r at time t'

$$alloc^{t'}_{(i,r)} = \frac{N^{t'}_{(i,r)}}{N^{t'}} * m$$

This formulation meets our objective of fairness across processes as illustrated in Section 3.4.

The above formulation ensures that resources are divided fairly across processes and regions. However, there may be applications in which the processes must be executed in a predetermined order. To address this class of applications, we allow for the specification of a process priority in the application description. To accommodate these requirements, we first sort the queue used for drone allocation by process priority and then by service level. In a resource constrained situation, we allocate drones to the higher priority processes, allowing lower priority processes to starve.

4.3 Dispatcher

The Dispatcher is responsible for carrying out the allocation decisions made by the Scheduler. Specifically, it manages the drone inventory and prepares drones for sorties by programming the specified behavior onto the drone and parameterizing the starting region. When a drone returns, the Dispatcher invokes a process to merge the drone's scratchpad with the hive Datastore and initiates a charge cycle. When drones are fully charged and ready to be dispatched, the Dispatcher notifies the Controller of the resource availability.

In Section 3, we define the term information latency as the difference in time between an event occurring in the world and the hive being informed of the occurrence. This latency is a function of the swarm deployment (e.g. swarm size, sortie time, charge time, maximum velocity) and dispatch policy. In general, it is desirable to minimize the information latency in the system, but it is particularly important for applications that track highly dynamic or continuous phenomena. In Karma we manipulate the dispatch policy to better fit the application and minimize the information latency. The Controller in Karma operates in two phases; it first determines how many resources each process-region pair should be granted by invoking the Scheduler, and then dispatches drones (using the Dispatcher) to fill the allocation requests. The algorithm for determining the amount of resources to grant is fixed, but the dispatch policy may vary. As such, we propose two dispatch policies. The goal of the *continuous dispatch policy* in Karma is to ensure a constant presence of drones in the field and minimize the information latency in the system by amortizing the total allocation of drones to a region over a period of time (sortie time + charge time). In contrast, the *greedy dispatch policy* dispatches drones opportunistically. We investiagate the effects of these policies in our system evaluation.

Reducing information latency can have a significant impact on the quality of data collected by applications that track continuously changing phenomena, such as chemical plumes. The application programmer should decide how information latency affects the application at hand. We allow the application programmer to add this as part of the program specification, and the corresponding dispatch policy is selected accordingly.

```
process Search
  runs RandomWalkSearch
  uses ()
  yields ('obs', 'feature')
  activated when ('obs' < 250)
  progress := ('obs' / 250)

process Survey
  runs RandomWalkSurvey
  uses ('feature')
  yields ('studied')
  activated when ('feature' > 0)
  progress := (if isNull('feature')
               then 1
               else ('studied' / (2 * 'feature')))
```

Figure 5. Psuedo-code for the definition of processes that make up the walkthrough application.

4.4 Execution Walkthrough

We conclude the description of the system by walking through an execution trace of a simplified application. We illustrate key features of the system by examining the decisions made by the Controller over the course of the execution. The example application consists of two processes, `Search` and `Survey`. Process `Search` runs a behavior executed uniformly over the target area and produces information relating to interesting features observed in the field. Process `Survey` runs a behavior that is triggered to execute only in areas identified as interesting by the `Search` process. The pseudo-code for this application is given in Figure 5.

We execute this application in simulation using 200 drones in a target area that is 75 x 75 meters. The world is partitioned into regions by dividing the target area as a grid with 10 rows and 10 columns. The hive is placed at the center of this area. A circular area representing the presence of interesting features is modeled with its origin at $(15, 15)$ and a radius of 10 meters. For simplicity, no weather or hardware failure is modeled. A greedy dispatch policy is used on the hive. Each drone is given enough energy to complete a 40-50 second sortie, with a subsequent charge time of about 2 minutes.

Figure 6 depicts an execution trace for this application. The top panel shows the amount of remaining work over time. The bottom panel shows the number of drones that are allocated to each process over time. This data illustrates two points about our scheduling algorithm. First, the drones appear to be allocated in waves. For the most part, this is true. The peaks in the bottom panel represent times when drones were dispatched (opportunistically using the greedy policy) and the valleys represent charging cycles. Since the drones are returning from different regions in the target area and have slightly different battery capacities upon returning, they are not available for scheduling at exactly the same time. This creates the jagged appearance of each "step" in the allocation plot. Second, notice that the `Survey` process is initially idle, and remains idle until the first `Search` sortie returns to the hive and the drones deposit the collected information into the central Datastore, demonstrating the implicit dependency between the two processes. Third, the bootstrapping sortie, as described in Section 4.2, is required when there is a lack of information about the progress rate for a process in a given region. In this example the number of drones allocated to the first sortie of each process is limited to one per region (100 total) despite the fact that more drones are available at the outset. Finally, the number of drones allocated to each process is proportional to the amount of work remaining and is a function of the estimated progress rate. For most of the execution the amount of work remaining for `Survey` is far below that of `Search`, so `Search` is allocated most of the resources.

Figure 7 illustrates how the dependency between processes results in selective allocation. The lefthand panel shows a drawing of the world in which the application is running. The hive is at the center and the world and an area containing interesting features is set in the top right as shown. The `Search` process records observations of these features when it executes in this area. The scheduler evaluates the activation predicate for the `Survey` process in each region of the discretized world and allocates drones accordingly. The righthand panel depicts the resulting cumulative allocation of drones to regions for the `Survey` process over the course of the execution. Without prior knowledge, our system correctly allocates drones to execute the `Survey` behavior only in regions where this behavior is useful. This allocation falls out of a single data dependency in the activation predicate.

This example demonstrates how the key features of Karma, progress rate estimation, proportional scheduling, and selective activation of processes in regions, can be combined to execute a swarm application.

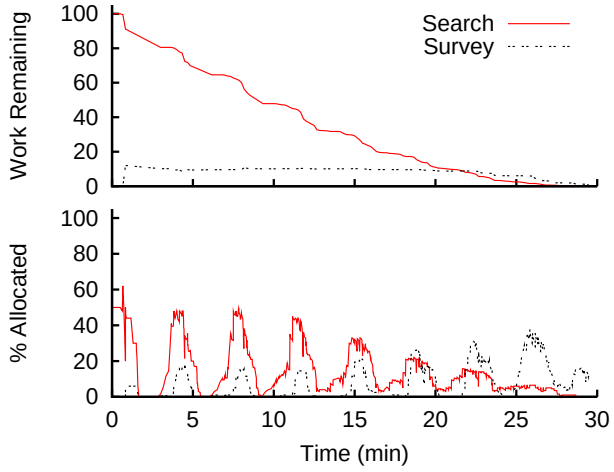


Figure 6. Karma allocates drones to behaviors according to the estimated amount of work to be done and the measured progress rate of each behavior per region. Remaining work is the sum of remaining progress across all regions.

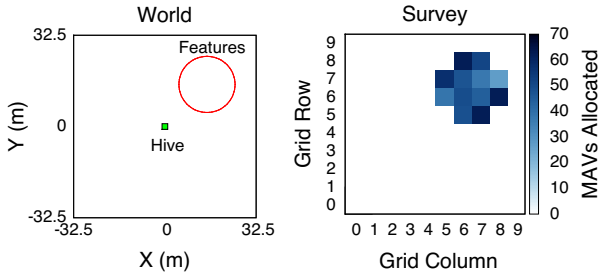


Figure 7. Processes are selectively activated by the presence or absence of information. The righthand panel shows the regions in which the `Survey` process is activated by the prior detection of environmental features.

5 Evaluation

In this section we demonstrate that the Karma system can be used to manage a swarm of MAVs and effectively execute applications inspired by real world workloads. We characterize the effectiveness of our system by evaluating its performance with respect to three metrics; *execution time, energy cost*, and *information latency*. Completion time and energy cost are useful metrics of efficiency. For instance, a farmer may want to minimize the total execution time so that a single hive can be shared among a number of fields on a fixed schedule. However, minimizing completion time may result in more resources being consumed than is strictly necessary, forcing a tradeoff to be made. We evaluate the scheduling decisions made by the system in the context of this tradeoff. Increased information latency is a direct result of the hive-drone paradigm. It is especially problematic for applications that track features of the environment that change frequently or continuously. To this end, we use this metric to evaluate how the selection of a dispatch policy can mitigate the negative effects of the sortie model.

5.1 Simulation Setup

We have developed a simulation environment, *Simbeeotic*, that allows us to test our ideas in a realistic, three-dimensional virtual world. The simulator is written in Java and builds atop an open source, six degrees of freedom, rigid body physics engine, JBullet.[2] JBullet provides realistic physical interaction between objects in the virtual world. Programmers write routines for controlling actuation and attach them to virtual MAV platforms, which can be equipped with a number of virtual sensors (e.g. accelerometer, gyroscope, range finder, compass, optical flow). In addition, users can create virtual worlds with flower patches, buildings, obstacles, and environmental effects such as gravity and wind. With this tool we are able to simulate swarms of MAVs at scales well beyond the practical limitations of our prototype testbed.

The Karma implementation consists of two runtimes; one that executes on the hive, and one that is embedded on the drones. The hive runtime is responsible for monitoring the progress of applications, scheduling and dispatching sorties, and charging drones. The drone runtime provides location and data storage services to running behaviors. The Karma hive runtime is implemented as a standalone Java application and is decoupled from the drone runtimes by a custom *dispatch driver*. This arrangement allows the Karma hive to execute the same application on multiple deployments. Our evaluation is based on experiments carried out in Simbeeotic and on a testbed of toy helicopters using this mechanism. All of the results shown are obtained in simulation, with the exception of the testbed experiment in Section 5.6.

We evaluate our system by using the alfalfa crop monitoring and pollination application introduced in Section 2.1 and depicted in Figure 4. In the following experiments, we run this application in a model alfalfa field that is one acre in area (63.63 by 63.63 meters) discretized into a 6x6 region grid. The origin of the world is at the center of the field and the hive is placed outside the field at $(-35, 0)$. The application is designed to execute for an entire growing season, periodically re-executing the three monitoring behaviors and activating the pollination behavior when the crop is in bloom. As written, the application expects the field to be monitored for pests, disease, and flower blooms *once per day*. Our experiments consist of single day snapshots of the application rather than the entire season. The battery and energy model of the drones are parameterized to provide approximate sortie and charge times of 5 minutes and 20 minutes respectively. The drones have a top speed of 2 meters per second when cruising to and from the hive, but operate at speeds between 0.25 and 0.5 meters per second when executing behaviors. The Scheduler is set to allocate drones (if available) every 10 seconds. Statistics for experimental results are gathered by repeating simulated experiments five times with varying random seeds. Unless otherwise noted, a swarm size of 800 drones is used in combination with a greedy dispatch policy for all experiments.

[2] `http://jbullet.advel.cz`

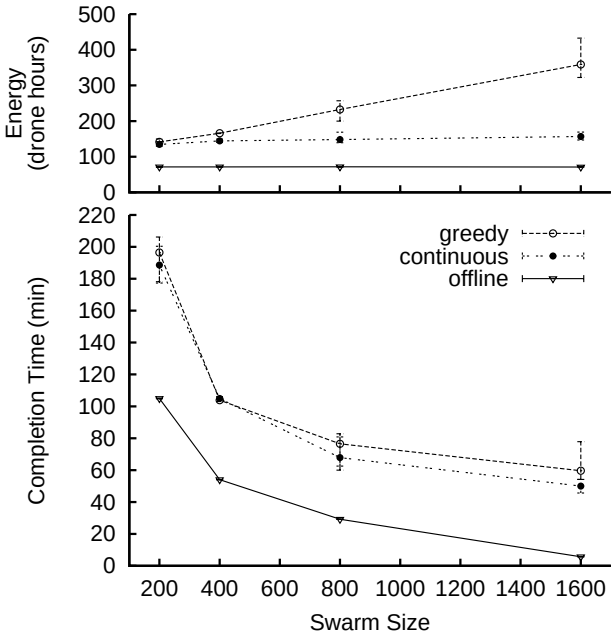


Figure 8. Karma scales sub-linearly (w.r.t. completion time) as swarm size increases. Gains are offset by relatively long charging periods in short scenarios.

5.2 Efficiency

We evaluate the efficiency of our system by comparing the overall completion time and energy cost of the alfalfa crop monitoring and pollination application as executed by Karma in Simbeeotic to a restricted version of an oracle offline scheduling model. For a fair comparison, the offline solution is required to use the gridded spatial decomposition and search each region for flower blooms despite having perfect knowledge of the bloom density. For each drone scheduled, the oracle model has full knowledge of the performance of all previously deployed drones. Since drones are scheduled concurrently, this implies that the oracle model has *foreknowledge* of all the activity of the drones that are currently deployed. As such, the Scheduler can allocate the minimum number of drones required to complete the application without having to estimate drone performance. We use this oracle model as a theoretical lower bound for comparison with Karma, even though it is not attainable in practice.

Figure 8 shows the scenario completion time and expended energy as a function of swarm size. We do not expect the performance (w.r.t. completion time) of Karma to improve linearly with swarm size in short scenarios because recharging is the dominant contributor to swarm overhead. Unless there is enough swarm growth to reduce the number of sorties executed by each drone, the gains will be limited. As expected, the oracle solution scales nearly linearly to the point where there are enough drones to complete all of the required work without recharging (about 850 drones). Karma scales sub-linearly with swarm size. The performance gap between Karma and the offline solution is mainly attributed to the bootstrapping sorties (with the accompanying charge period) and the information latency caused by muling data.

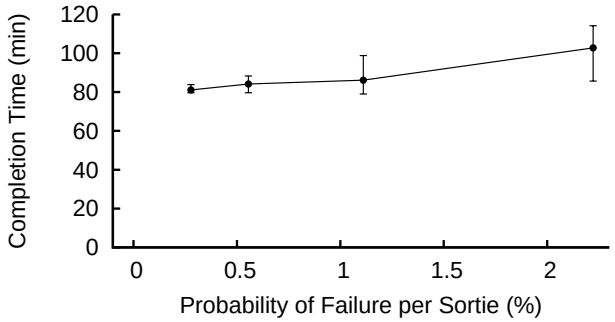


Figure 9. Karma is resilient to individual drone failures, exhibiting a graceful degradation in performance as the probability of failure increases.

We evaluate Karma using the greedy and continuous dispatch policies described in Section 4.3. Since the workload is fixed and the evaluation metric is total completion time, the performance of the two dispatch policies is roughly equivalent. However, there is a disparity between the two dispatch policies with respect to energy consumption. As the size of the swarm increases, the system using the greedy dispatch policy consumes 50% more resources. This is due to a combination of poor allocation estimates and opportunistic dispatching. The greedy dispatch policy tends to release drones in batches. Empirical evidence suggests that early allocation estimates are inaccurate, which leads to an overcommitment of resources. Since the continuous policy staggers drone dispatching over time, the erroneous resource allocations can be gradually corrected with limited overhead.

5.3 Resilience to Failure

We demonstrate Karma's resilience to individual drone failure. Our system is designed to provide a graceful degradation in performance (as measured by overall completion time) that is proportional to the amount of failure that occurs. The approach we take follows naturally from the hive-drone model and the use of progress rates to estimate resource allocation needs. When a drone fails in the field, it never returns to update the Datastore and, from the perspective of the hive, no progress is made. The Dispatcher will detect the failure with a timeout and inform the Scheduler that it has one less resource in the field and that the swarm size has decreased. The Scheduler will take this into account during the next allocation cycle and update its estimates accordingly. This mechanism works well when failure is uniformly distributed across all regions of the target area. However, there is a corner case in which a disturbance (e.g. strong wind) is localized to a subset of the area. In this case we would prefer that the system detect the anomaly and discontinue sending drones to the hazardous area. Because this disturbance may also be time-varying, an attempt could be made to resume allocation to the problem region after a period of time. Handling these situations in Karma is left as future work.

We evaluate the effectiveness of our system design with respect to failure by executing the alfalfa crop monitoring and pollination application with a swarm of drones that have a constant probability of failure. With this model we expect to see an increase in the total execution time of the application with an increase in failure. As shown in Figure 9, Karma handles the unexpected failures with a predicted graceful degradation in performance. The majority of the extra time is due to the swarm's inability to immediately react to a detected failure (all of the drones are deployed or charging). This issue can be mitigated by reserving drones or increasing the swarm size. In addition to the time overhead, there is a small energy penalty (5% in the worst case scenario) that is caused by the additional sorties required to make up for the lost drones. The swarm size may dwindle to the point that no progress can be made, but that point is reached through graceful degradation, not hard failure.

5.4 Adaptability

Consider the impact on a drone of wind blowing at a constant speed and direction. The drone must work harder to compensate for the additional force and avoid being blown off-course. As a result, it will expend more energy to fight the wind, resulting in a shorter sortie time. In turn, the progress rate for the behavior it is executing will be reduced and the system will produce higher resource estimates.

In the following experiment we modify the alfalfa crop monitoring and pollination application to introduce a constant wind over the bottom third of the field. Drones dispatched to the bottom third of the field experienced a 32% reduction in sortie time. Because the workload is constant for the four behaviors in this application, shorter sortie times result in less work being accomplished per sortie, which reduces the progress rate for these regions. In this case, the Scheduler responds by allocating 12% more drones than the wind-free regions to accomplish the same amount of work. Correspondingly, the energy cost for executing in the presence of regional wind is 7% higher than that of the windless scenario.

The environmental dynamics are implicitly captured in the regional behavior progress rates. Even though the Karma system did not explicitly measure the wind speed, the swarm was able to adapt to its presence. By representing the dynamics in a single variable (the progress rate), the hive is able to adapt to external influences, but it cannot disambiguate the causes or apply specific solutions. Though this experiment demonstrates spatial variation, the dynamics of the environment can also change over time. The Scheduler's use of the regional progress rates to capture these dynamics also accounts for temporal fluctuation. The accuracy with which the scheduler can track the progress rate (and implicitly the environmental dynamics influencing it) depends on how frequently that rate is sampled (how often a drone returns from that region). This is defined in Section 3 as the information latency problem, and is addressed in the next set of experiments.

5.5 Information Latency

The previous experiment focused on evaluating our system in the presence of environmental dynamics. However, the features of interest that the application is interested in detecting and tracking are essentially static (or do not change in some detectable way) over the period of execution. How does the system behave when applications aim to track phe-

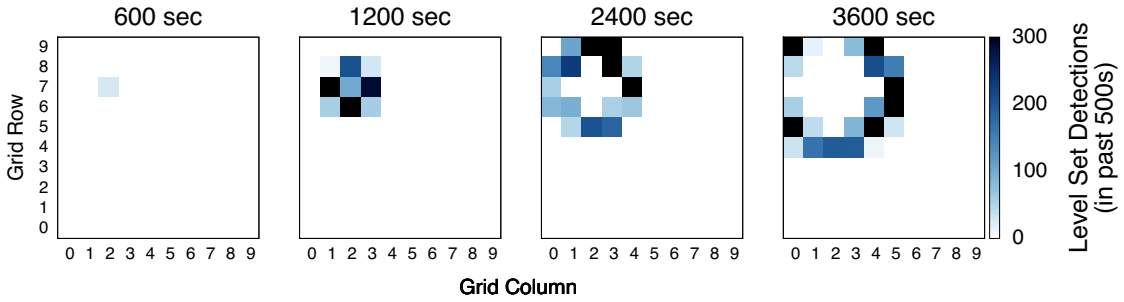


Figure 11. A series of snapshots from the hive Datastore depicting the measured contour of an expanding chemical plume.

```
process DetectPlume
  priority = 2
  runs RandomWalkPlumeSearch
  uses ()
  yields ('plume_obs', 'plume_det')
  activated when ('plume_obs':5m < 400)
  progress := ('plume_obs':5m / 400)

process FindLevelSet
  priority = 1
  runs RandomWalkLevelSet
  uses ('plume_det')
  yields ('level_obs', 'level_det')
  activated when ('plume_det':5m > 0)
  progress := (if (isNull('plume_det':5m)
               then 1
               else ('level_obs':5m / 500)))

process FindCenter
  priority = 1
  ...
```

Figure 10. Psuedo-code for the definition of processes that make up the plume tracking application.

nomena that change continuously? We demonstrate that the hive-drone paradigm can be used to continuously measure time varying phenomena. To this end, we define a chemical plume tracking application that consists of three behaviors; one to perform a uniform search of the target area, one to detect a level set (contour) of the plume, and one to find the center. Figure 10 defines this application in pseudo-code. The application is executed in a target area that is 100 meters by 100 meters, discretized into a 10x10 region grid. The plume is centered at $(-25, 25)$ and expands in a hemispherical pattern at a rate of 1 centimeter per second. Unless otherwise noted, a swarm size of 800 drones is used along with the continuous dispatch policy.

All three processes defined by the plume tracking application are unbounded, meaning that they are not meant to terminate after some fixed amount of work is done. Rather, they define sliding windows in which the activation predicates and progress functions are evaluated as time moves forward. Using these windows, drones are allocated to regions that have the most recent information, allowing the frontier of swarm activity to follow the plume as it expands. Figure 11 depicts a series of snapshots taken from the Datastore

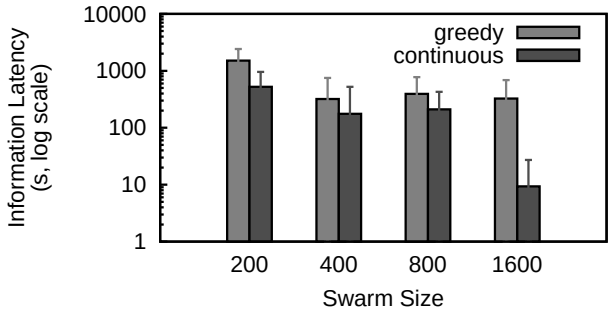


Figure 12. Information latency measurements for one process-region pair in the plume tracking application. The continuous dispatch policy consistently outperforms the greedy policy.

as the plume tracking application is executed. Using a windowed query, we are able define the regions in which the *level_det* feature was most recently observed. Older observations of this feature are stored in the Datastore, but are not included in the view defined by the windowed query.

In the next experiment, we quantify the effect that the dispatching policy has on information latency. We execute the plume tracking scenario with the two dispatch policies (greedy and continuous). As a proxy for information latency (as defined in Section 3), we measure the *process-region return period*; the amount of time between consecutive drones returning from each region for each process. This allows us to measure how frequently we receive information from a region, which is directly related to the information latency on events that occur in that region. Because the drones that are dispatched concurrently do not return at exactly the same time, we cluster return events that occur within a 30 second period as one event.

Figure 12 shows the results of the experiment running for 6 virtual hours. Depicted is the mean and standard deviation of information latency measurements for a single process-region pair. As expected, the continuous policy outperforms the greedy policy with respect to minimizing information latency. As the swarm size increases, there are more drones available when the Scheduler requests resources and the charge time plays a smaller role in defining the information latency. On average, switching policies reduces the measured information latency by 63%, with an order of

Figure 13. The ground vehicle and helicopters operating in the indoor testbed.

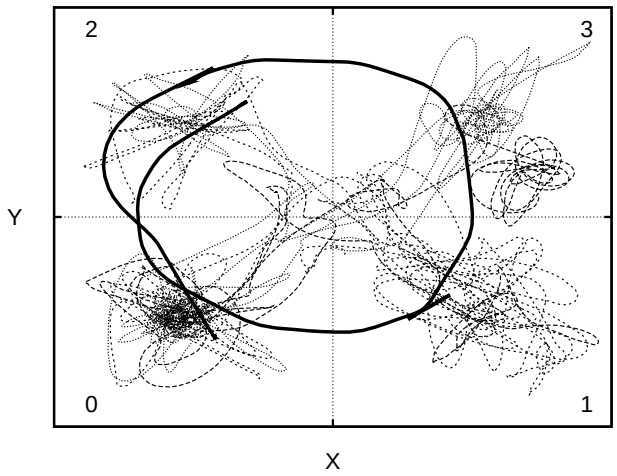


Figure 14. Ground truth target location and helicopter flight paths as recorded by the motion capture system during the testbed experiment. The target is solid and the helicopters are dotted.

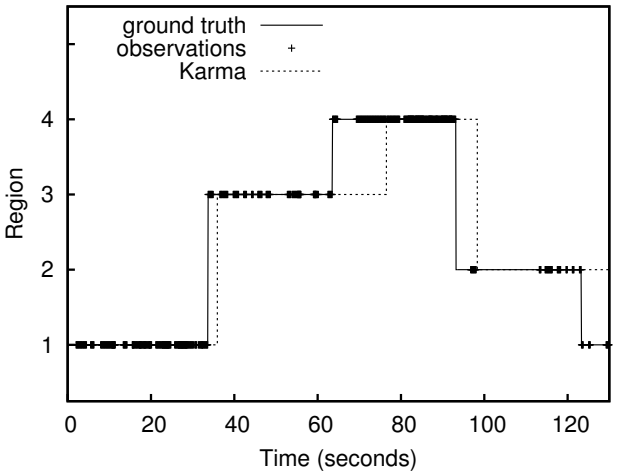


Figure 15. The region occupied by the target throughout the experiment. The perceived location (dotted line) lags behind the truth location due to information latency.

magnitude improvement (97%) when the swarm size is increased to 1600 drones. The takeaway is that certain aspects of the Karma system are coupled with the application, and customizing the system deployment can have a significant impact on the application performance.

5.6 Helicopter Testbed

In addition to the experiments in simulation, we evaluate Karma on a prototype MAV testbed. Our prototype MAV is the low cost ($120), E-flite Blade mCX2 micro coaxial radio helicopter. The mCX2 is 20 cm long, 12 cm high, weighs 28 grams, and has a flight time of 6–8 minutes depending on flight maneuvers.

We have integrated the mCX2 helicopter with Simbeeotic such that the helicopter can be controlled by behaviors running in the simulation. This hardware-in-the-loop setup allows us to control a model of the helicopter in the virtual world (using virtual sensors), while the helicopter flies in the physical world. We modify the mCX2 radio transmitter to accept commands from a USB port, allowing us to send RC commands to the helicopter when its model is controlled. To complete the loop, the physical position and orientation of the helicopter is captured by a precision Vicon[3] motion capture system and injected into the model's state. Note that no Karma code is executing onboard the helicopter, though this is the subject of ongoing work.

For our testbed experiment, we evaluate a simple tracking application using three helicopters and one ground vehicle (shown in Figure 13). Our testbed space is divided into four equally sized regions. Initially, Karma dispatches helicopters with the `Search` behavior, causing the helicopters to look for ground vehicles. When a helicopter has located a vehicle, it writes the location of the vehicle to its local scratchpad. When the drone returns to the hive and uploads its data, the tracking process is activated. Karma then dispatches a helicopter with the `Track` behavior to the last known vehicle location to resume observation.

The ground vehicle starts off stationary in the center of one of the regions. After approximately 30 seconds the vehicle moves to an adjacent region and waits another 30 seconds before moving on. This process repeats for the duration of the experiment. The helicopters fly 20 seconds sorties. This is short of the maximum flight time of the helicopter, but allows us to see Karma perform several sorties during the course of the experiment. We equip the helicopters with a virtual sensor that allows them to locate the ground vehicle if they are within 90cm in the x-y plane.

Figure 14 shows a ground truth trace of the tracking experiment. At the start, the vehicle begins in region 1 and moves in a clockwise fashion. Figure 15 shows a region-level trace of the target location during the experiment. The markers indicate when a helicopter records an observation into its scratchpad. The hive-drone model introduces information latency, which can be seen here as a gap (on the time axis) between the solid and dotted lines, which represent the truth and perceived target location, respectively.

[3] `http://www.vicon.com`

6 Discussion

We demonstrate that the Karma system is capable of executing swarm applications with reasonable efficiency and resiliency. However, there are some aspects of the system design that warrant further discussion and study.

In our design, each drone is assigned a behavior to execute per sortie. This policy is a direct result of the short flight times of the current MAV prototype. If the drones had longer flight times, multi-tasking on individual drones might allow for more efficient operation. Further, we only consider sorties that keep the drones deployed for the maximum amount of time. It may be possible to reduce information latency and gain more flexibility in scheduling by considering variable length sorties. In addition, a larger area could be covered if a multi-hive solution were adopted (with inter-hive communication), allowing for one-way sorties that redistribute the swarm's resources at runtime. These features would require significant modifications to the Scheduler to incorporate resource planning.

The Karma scheduler relies on estimating the progress rates of drones executing behaviors in the field. Though this allows the system to adapt to varying conditions, it can be problematic when the estimate is inaccurate. It may be possible to produce more robust estimates by incorporating values from neighboring regions or using a priori models. An over-provisioning strategy could mitigate the impact of overestimation (or underperforming drones) on completion time at an additional cost in resources.

7 Related Work

Programming clusters of computers is a long-standing problem that has received ample attention in research. River [1] provides adaptive mechanisms that allow database query-processing applications to cope with performance variations in cluster platforms. They propose a dataflow programming model and two new constructs – a distributed queue to decouple the dependence of producers of data from consumers and a graduated declustering mechanism which decouples the consumers from producers. Like Karma, River handles massively parallel operations by estimating the performance variations and doing intelligent scheduling to achieve the best performance. However, the workloads are inherently parallel in River whereas the parallelism is achieved in partitioning space and simplifying the sequence of actions in a behavior in Karma. Applications in Dryad [10] and CIEL [17] are expressed as dataflow graphs. The Karma programming model avoids the explicit parallelization used in Dryad graphs, allowing processes to be scaled with the size of the swarm. Although the process interdependencies in Karma applications can be similarly expressed in the dataflow graphs of cluster computing systems, the scheduling problem is quite different. Cluster systems like MapReduce and Dryad must schedule jobs while optimizing for data locality, network usage, and resource availability. However, the inputs and outputs of these systems are mostly deterministic (a static partitioning of existing data), whereas the operation of a MAV swarm in an unknown environment is anything but predictable. This is why Karma takes a reactive approach to scheduling the system resources, basing decisions on feedback. CIEL provides support for iterative and recursive computations by allowing the the dataflow graph to be modified at runtime. We believe this programming model is well suited for MAV swarms, and hope to explore its usefulness as a frontend to Karma in future work.

Spatially-oriented computing offers an alternative approach to procedural multi-robot programming. In this paradigm, space is used as a first-class computing abstraction, and individual nodes typically act according to some spatially-oriented conditions. Protoswarm [3] is a language that presents the swarm as a single continuous spatial computer. Meld [2] is a declarative logic-programming language to program robotic ensembles. Meld was designed for modular robots where the inter-robot communication is limited to immediate neighbors. Locally distributed predicates [5] are distributed conditions that hold for a connected ensemble of the robotic system. Programs in this paradigm are collection of LDPs with actions that are triggered when subensembles match a particular predicate. Karma partitions space to achieve data-parallel operation.

Swarm robotics and swarm intelligence research applies resource management techniques at large scale. Swarm robotics algorithms and systems typically focus on emergent behavior arising from local decisions made by large numbers of simple agents. These often biologically-inspired algorithms have proven successful in diverse tasks such as collective construction [23] and multi-parameter optimization [12]. The large body of work in this area is directly applicable to MAV swarm programming. In designing Karma, we have eschewed the principles of emergent collective behavior in favor of explicit, global coordination based on the hive-drone model. We employ simple agent behaviors, and move most of the complexity to the central hive. Since resource allocation is an iterative, centralized problem, we are able to make a reasoned assessment of swarm progress – something that is often difficult with emergent algorithms.

8 Conclusions

MAV swarms are an emerging class of mobile sensing systems. However, challenges exist in programming such systems. We propose a novel system architecture based on the hive-drone model. The model uses a programming abstraction that simplifies programming individual MAVs and shifts the coordination complexity to a central hive. We implement this model in our prototype system called Karma and show that it is efficient, adaptive, and resilient to failure.

9 Acknowledgements

We would like to thank to our shepherd, Philip Levis, Geoffrey Challen, and the anonymous reviewers for their insight and detailed feedback. Special thanks to Margo Seltzer for her participation and thoughtful commentary that vastly improved the quality of this work. This work was partially supported by the National Science Foundation (award number CCF-0926148). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10 References

[1] R. H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Transactions on Computer Systems (TOCS)*, 21(1):36–86, February 2003.

[2] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2007.

[3] J. Bachrach, J. McLurkin, and A. Grue. Protoswarm: a language for programming multi-robot systems using the amorphous medium abstraction. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1175–1178, 2008.

[4] G. L. Barrows, J. S. Chahl, and Y. V. Srinivasan. Biomimetic visual sensing and flight control. In *Proceedings of the Bristol UAV Conference*, pages 159–168, 2002.

[5] M. De Rosa, S. Goldstein, P. Lee, P. Pillai, and J. Campbell. Programming modular robots with locally distributed predicates. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3156–3162, May 2008.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[7] K. S. Delaplane and D. F. Mayer. *Crop Pollination by Bees*. CABI Publishing, New York, NY, 2000.

[8] X. Deng, L. Schenato, W. C. Wu, and S. Sastry. Flapping flight for biomimetic robotic insects: Part i-system modeling. *IEEE Transactions on Robotics*, 22(4):776 –788, August 2006.

[9] M. H. Dickinson, F.-O. Lehmann, and S. P. Sane. Wing rotation and the aerodynamic basis of insect flight. *Science*, 284(5422):1954–1960, 1999.

[10] M. Isard, M. Budiu, Y. Yu, and A. Birrell. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[11] M. Karpelson, J. Whitney, G.-Y. Wei, and R.J.Wood. Energetics of flapping-wing robotic insects: Towards autonomous hovering flight. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2010.

[12] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, 1995.

[13] S. Koppal, I. Gkioulekas, G. Barrows, and T. Zickler. Wide-angle micro sensors for vision on a tight budget. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2011.

[14] B.-K. Lai, K. Kerman, and S. Ramanathan. Nanostructured $La_{0.6}Sr_{0.4}Co_{0.8}Fe_{0.2}O_3$ / $Y_{0.08}Zr_{0.92}O_{1.96}$ / $La_{0.6}Sr_{0.4}Co_{0.8}Fe_{0.2}O_3$ (LSCF/YSZ/LSCF) symmetric thin film solid oxide fuel cells. *Journal of Power Sources*, 196(4):1826 – 1832, 2011.

[15] D. Lambrinos, R. Moller, T. Labhart, and R. Pfeifer. A mobile robot employing insect strategies for navigation. *Robotics and Autonomous Systems*, 30(1-2):39–64, 2000.

[16] J. W. Langelaan and N. Roy. Enabling New Missions for Robotic Aircraft. *Science*, 2009.

[17] D. Murray, M. Schwarzkopf, and C. Smowton. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[18] D. Psychoudakis, W. Moulder, C.-C. Chen, H. Zhu, and J. L. Volakis. A portable low-power harmonic radar system and conformal tag for insect tracking. *IEEE Antennas and Wireless Propagation Letters*, 7:444–447, 2008.

[19] A. Purohit, Z. Sun, M. Salas, and P. Zhang. SensorFly: Controlled-mobile Sensing Platform for Indoor Emergency Response Applications. In *Proceedings of the 10th International Conference on Information Processing in Sensor Networks (IPSN)*, 2011.

[20] P. Sreetharan and R. Wood. Passive torque regulation in an underactuated flapping wing robotic insect. In *Proceedings of Robotics: Science and Systems*, June 2010.

[21] E. Stokstad. The Case of the Empty Hives. *Science*, 316(5827):970–972, 2007.

[22] H. Tanaka and R. J. Wood. Fabrication of corrugated artificial insect wings using laser micromachined molds. *Journal of Micromechanics and Microengineering*, 20(7):075008, 2010.

[23] G. Theraulaz and E. Bonabeau. Coordination in distributed building. *Science*, 269(5224):686–688, 1995.

[24] R. J. Wood. The first takeoff of a biologically inspired at-scale robotic insect. *IEEE Transactions on Robotics*, 24(2):341 –347, April 2008.