Using Game Theory to Manage Self-Aware Unmanned Aerial Systems

Venkata Mandadapu and Christopher Stewart
The Ohio State University

***Abstract*— Runtime platforms on unmanned aerial systems (UAS) manage flight, GPS and compute resources and speed up common tasks for UAS workloads. These workloads evolve rapidly due to programmer demand and changing external conditions. Platforms are quickly out dated. Existing self-aware techniques update resource management policies and/or software, but managing the cost of updates under a budget is challenging. This paper makes the case for using game theory. Our approach profiles currently hosted workloads and measures efficiency gains from updates. Counter-factual regret, a game theory technique, computes when platforms should update. We outline a framework, provide an example and discuss research challenges.**

I. Introduction

Da-Jiang Innovations (DJI), the largest producer of commercial drones, now supports programmatic access and control of unmanned aerial systems (UAS) before and during flight. UAS include a wide range of resources, including: processors, wireless cards, flash storage, cameras, gimbals, GPS devices, batteries and, of course, rotors, wings and motors. The DJI SDK, available for Android, MacOS and Linux operating systems, provides an API to access these resources.PixHawk and Parrot also provide similar runtime platforms.

Programmable runtime platforms qualitatively change drone workloads. Instead of requiring a human to manage a remote control and maintain line of sight, programmable drones can (1) fly to specific GPS locations and return home, (2) execute complex computations on-the-spot and (3) change their flight plans based on dynamic conditions. Runtime platforms and the applications they enable will soon drive the drone market. As shown in Figure 1 sales for unmanned aerial systems have outpaced sales for their underlying components (i.e., image sensors for cameras and altimeters for flight). Programmers are increasingly excited about using UAS runtime platforms. In 2016, the DJI Developer Competition attracted over 130 developers worldwide. Programmers often foretell sales growth. Figure 2 plots the percentage gain in programmers excited about Android, IOS or Windows smart phone platforms against annual platform sales. The R^2 correlation coefficient is 0.87 [12].

In addition to programmer demand, novel execution contexts affect UAS workloads. For example, small UAS allow users to fly indoors and collect a new class of sensed images. Improved energy efficiency allows UAS to execute missions in remote areas without human intervention.

With rapid workload change, UAS platforms must frequently update resource management policies. Research on self-adaptive systems has addressed the technical challenges

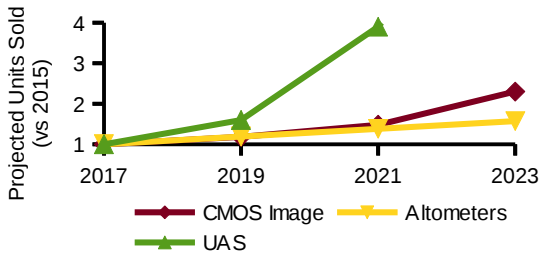

Fig. 1. Sales for drones vs constituent components.

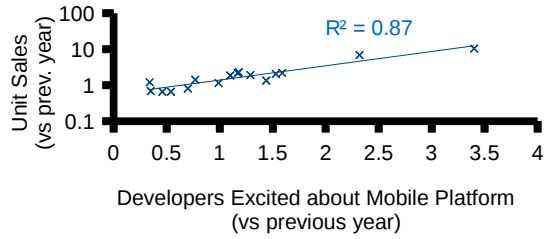

Fig. 2. Developer excitement correlates with sales.

of (1) setting goals for resource management, especially in complex systems [2], [8], (2) monitoring policies and assessing efficiency [3], [7], (3) choosing between competing policies and models [17] and (3) efficiently changing policies on the fly [4], [5], [9], [16]. However, these approaches often exclude the cost of updating platforms. With rapid workload change, the costs can become prohibitive.

This position paper addresses a problem created by the unprecedented growth of programmable UAS: *How often should runtime platforms update policies and/or software given limited resources for such updates?* As a running example, we ask the reader to consider software updates, because they require costly programming effort. We argue that game theory approaches can help. Specifically, we propose a framework with the following steps:

1. **Profile platforms over time:** For resource management policies, profiles will assess runtime metrics, e.g., energy efficiency or response time. For software updates, profiles will assess code. Later, we speedup from API packages. Profiles capture external demand shifts, following self-adaptive practices [7], [13], [14], [18].

2. **Develop antagonists:** These agents have constrained abilities to affect demand but aim to degrade platform efficiency. These agents provide worst-case bounds the cost of updates. They can learn static update policies and tailor the impact on demand for maximum effect.

3. **Use counterfactual regret:** Counter-factual regret is an iterative procedure to minimize the effect of antagonistic forces. In the context of this paper, the approach first sets update policies, trains antagonists to mitigate the policy and then updates policies again. The approach continues until convergence.

Counter-factual regret has been applied famously to solve Limit Hold'em Poker [1]. A key position of this paper is that poker and platform management are similar problems. Both suffer from stochastic external forces (shuffled decks vs changing demand). Antagonists are explicit in poker. In platform management, they are implicit, emerging from the conservative desire to avoid exceeding cost budgets from updates. Effectively, antagonists allow platform developers to study realistic worst case costs of their update policies.

The remainder of this paper is as follows. Section II outlines counterfactual regret, an approach to manage decision making in stochastic games. Section III proposes a framework that updates UAS runtimes using counterfactual regret. Section IV provides a brief discussion on outstanding challenges.

II. Counterfactual Regret

Consider two agents that repeatedly compete for resources. Agents act in turns and their actions are influenced by data about prior actions and their effects. Over a long period, each agent would like to make choices that maximize their *utility*, where utility is a function of resources acquired. In this context, the Regret Matching Algorithm [1] maps recent actions and their stochastic effects to the next action an agent should take. The algorithm reaches equilibrium when mappings stabilize for each agent.

An agent's *regret* for an action taken is the difference between utility received and utility that could have been received with the best alternative action. If an agent takes the action with maximum utility then regret is zero. With Regret Matching, each agent minimizes its regret by taking different actions when faced with similar conditions. Each agent tracks past choices and makes decisions by choosing choices with least regret to improve the utility of the player.

Now, consider agents that must take multiple actions before knowing their utility. In this context, regret minimization is challenging because the space of possible action sequences is large. The *Counterfactual Regret Minimization* algorithm addresses this challenge [1], [19] by decomposing overall regret into multiple additive regret terms. Each player starts with a random strategy for making choices and over multiple iterations updates the strategy based marginal regret of each action. Figure 3 depicts this approach in the context of update policies.

After many iterations, each agent develops a strategy that converges to a Nash equilibrium [10]. That is, a state where agents can not gain utility by changing strategy alone. Once Nash equilibrium is reached in order to improve the utility there should be an external effect like a new player joining the game— or a change in the rules of play. An ϵ-nash equilibrium allows a bounded gain for changing strategies.

Origin of Antagonists: We speculate that antagonistic workload demands arise in complex systems of systems. Emergent behavior in interwoven systems can be catastrophic or common [15]. For example, symbiotic cognitive systems involve compute systems that interact using keyboards, language and inference with human systems [6]. A major emergent behavior could cause an unexpected financial loss, but common behaviors like the development of computer-oriented oratory accents are also likely. In both cases, system managers need models to explore realistically worst-case conditions for complex systems. It is our position that prior control-theoretic approaches do not go far enough [16].

III. Example: Software Updates

Problem Statement: Programmers demand high-level programming abstractions. Runtime platforms provide a subset of those abstractions, allowing programmers to invoke them with few lines of code. Programmers must implement the remaining abstractions themselves, requiring many lines of code.

Platform developers strive to provide abstractions that minimize lines of code for all programmers, but programmer demands change over time. We model two types of programmers:

1. **Typical programmers** shift their demands according to a known stochastic distribution over time. These shifts reflect predictable changes in the way technology will evolve, e.g., trends in processor speed and battery capacity.

2. **Antagonist programmers** shift their demands to make the platform inefficient. At most a fixed percentage of aggregate demand comes from this group, further game theoretic rules may govern this groups actions, constraining demand shifts from this group.

Game Theory: With this background, we define a game suitable for counterfactual regret. There are two agents: platform developers and antagonists. Periodically, platform developers can update their platform, aligning the API to programmer demands. Updates require costly human resources. We approximate these using the edit distance between two updated APIs. There is a global budget for updates over a long period (10 years). Between updates, typical and antagonist programmers shift demand for abstractions, either continuously or at discrete. Recall, typical users are stochastic whereas antagonists are strategic.

For platform developers, utility is the geometric mean of programmer efficiency over a long period (10 years). For antagonists, utility is the aggregate amount of code written. A Nash equilibrium is an update frequency where (1) program developers can not improve efficiency by increasing or

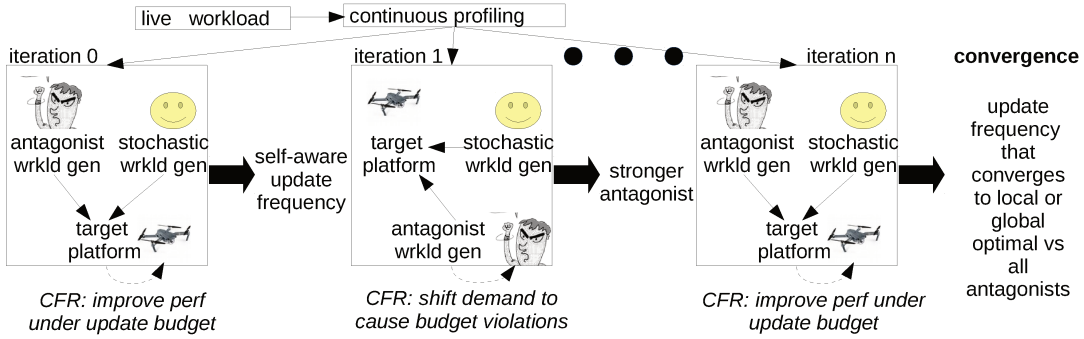

Fig. 3. Counter-factual regret involves thousands of simulations per iteration. In each iteration, the profiles informs aggregate demand and workload characteristics. Antagonists affect a portion of workload demand *under constraints*. During even iterations, antagonist behavior is fixed while optimizing platform. During odd iterations, platform update policies are fixed.

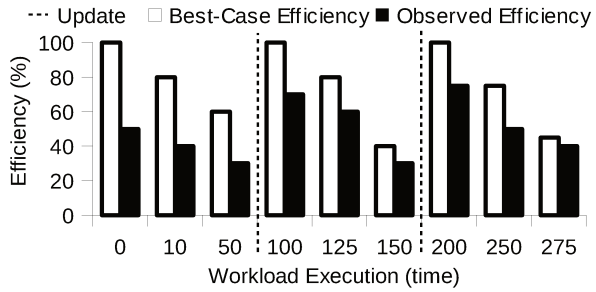

Fig. 4. A depiction of how the observed and best case efficiency change over time and how they compare against each other.

decreasing update rate and (2) antagonists can not force more lines of code.

Middleware Design: Our middleware platform manages communication on a UAS. Programmers use our middleware for commonly executed procedures instead of directly accessing UAS hardware or DJI interfaces. Our platform can aid a wide range of software programs from surveillance to agricultural crop scouting. The challenge is to monitor the efficiency of our middleware, detect how costly changes could be (in the worst case) and set an update frequency accordingly.

Our efficiency metric is shown below

$$\text{efficiency} = \frac{\text{LOC}_{\text{opt}}}{\text{LOC}_{\text{act}}} \tag{1}$$

where LOC_{opt} is lines of code in the version of application where middleware is used and LOC_{act} is lines of code where middleware is not used. As we can see in the Figure 4 as time proceeds the observed and best-case efficiency is reduced, when the gap is considerably small an update can be made to the platform. Observed efficiency is the efficiency of current platform where we try to provide the best implementation to most of the applications (generally the most used applications), best-case efficiency is the efficiency when the best implementation is provided to all the applications. There is difference in both of the efficiency metrics because the cost of update might not allow us to always provide best case implementations for all the applications.

The challenge is when should we update the platform. As time changes the capability of middleware to handle new applications and efficiency decreases which will reduce the value of middleware, but also updates cost so we cannot update the middleware everyday. So we have to find a perfect time when it is not too late and it is not too early to update. Function F(efficiency, t) gives the monetary benefit of updating the platform given the efficiency of the platform and time. Function G(updates) gives the cost of the update, the cost of the update is based on the number of lines that need to be written for the new update to be released.

$$\text{x} = \sum_{\bar{\text{t}}=\text{a}_\text{i}} \text{F}(\text{efficiency}, \bar{\text{t}}) \tag{2}$$

this equation gives us the total monetary benefit of all updates done to platform over a period of time.

$$\text{y} = \sum_{\forall \text{i}} \text{G}(\text{update}_\text{i}) \tag{3}$$

this equation gives us the total cost of all updates done to the platform. We would like maximize the sum of x and y.

We view the above maximization problem as a counterfactual regret minimization problem. The opponent in the design changes probability distribution of the usage of applications making the efficiency of the middleware to go down. The opponent will always try to change the distribution to increase usage of applications which are inefficient, but the opponent has restrictions on amount of change he can do in a fixed period of time. The regret for the opponent is the difference between the efficiency of platform for two different probability distributions.

$$\text{regret}_{\text{opp}} = \text{efficiency}_{\text{PD1}} - \text{efficiency}_{\text{PD2}} \tag{4}$$

The player is the developer of the platform, who creates new programs as the usage changes. The player will create programs in such way that the observed efficiency of the middleware is maximum. Regret for player is the difference in the efficiency of platform for two different implementations of programs.

$$\text{regret}_{\text{player}} = \text{efficiency}_{\text{I1}} - \text{efficiency}_{\text{I2}} \tag{5}$$

where I is the set of implementations of the programs.

Over course of time both the player and opponent will keep on making moves to reduce their regret until they reach the ϵ-nash equilibrium, when it is reached no one can make a move which will increase their utility by a large amount. This is the time when the platform should be updated. The update will take into consideration the probability distribution of usage of workload at that time and will change implementations of various workloads.

IV. Conclusion

Emerging UAS platforms face rapidly changing workload demands due to their popularity and sensitivity to execution context. Self-aware platforms can update resource management policies and software, but doing so under a budget is challenging. We have made a case for counterfactual regret, an agent-based game theoretic approach, as a tool to set effective update policies. Future work could include:

- Using counterfactual regret to manage realistic workloads. UAS need more, open platforms for such research [11].

- Applying counterfactual regrer to long running workloads that suffer black swan events.

- Rigorous proofs of convergence are needed for applications to UAS, IoT and self-driving cars where mistakes have grave consequences.

References

[1] M. Bowling, N. Burch, M. Johanson, and O. Tammelin. Heads-up limit holdem poker is solved. *Science*, 347(6218), 2015.

[2] E. Cavalcante, T. Batista, N. Bencomo, and P. Sawyer. Revisiting goal-oriented models for self-aware systems-of-systems. In *ICAC*, 2015.

[3] H. Hoffmann and M. Maggio. Pcp: A generalized approach to optimizing performance under power constraints through resource management. In *ICAC*, 2014.

[4] S. A. Javadi and A. Gandhi. Dial: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing. In *Autonomic Computing (ICAC), 2017 IEEE International Conference on*, pages 135–144. IEEE, 2017.

[5] J. Kelley, C. Stewart, N. Morris, D. Tiwari, Y. He, and S. Elnikety. Measuring and managing answer quality for online data-intensive services. In *ICAC*, 2015.

[6] J. O. Kephart and J. Lenchner. A symbiotic cognitive computing perspective on autonomic computing. In *ICAC*, 2015.

[7] S. Kounev, N. Huber, F. Brosig, and X. Zhu. A model-based approach to designing self-aware it systems and infrastructures. *Computer*, 49(7), 2016.

[8] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl. Efficient decision-making under uncertainty for proactive self-adaptation. In *ICAC*, 2016.

[9] N. Morris, S. M. Renganathan, C. Stewart, R. Birke, and L. Chen. Sprint ability: How well does your software exploit bursts in processing capacity? In *ICAC*, 2016.

[10] R. B. Myerson. Refinements of the nash equilibrium concept. *International journal of game theory*, 7(2), 1978.

[11] J. L. Sanchez-Lopez, R. A. S. Fernández, H. Bavle, C. Sampedro, M. Molina, J. Pestana, and P. Campoy. Aerostack: An architecture and open-source software framework for aerial robotics. In *International Conference on Unmanned Aircraft Systems*, 2016.

[12] Stackoverflow.com. Mobile developer surveys, 2013–2018.

[13] C. Stewart, M. Leventi, and K. Shen. Empirical examination of a collaborative web application. In *IISWC*, 2008.

[14] C. Stewart and K. Shen. Performance modeling and system management for multi-component online services. In *NSDI*, May 2005.

[15] S. Tomforde, S. Rudolph, K. L. Bellman, and R. P. Würtz. An organic computing perspective on self-improving system interweaving at runtime. In *ICAC*, 2016.

[16] Z. Wang, X. Liu, A. Zhang, C. Stewart, X. Zhu, and T. Kelly. Autoparam: Automated control of application-level performance in virtualized server environments. In *FeBid*, 2007.

[17] D. Weyns and M. U. Iftikhar. Model-based simulation at runtime for self-adaptive systems. In *ICAC*, 2016.

[18] Z. Xu, N. Deng, C. Stewart, and X. Wang. Cadre: Carbon-aware data replication for geo-diverse services. In *ICAC*, 2015.

[19] M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione. Regret minimization in games with incomplete information. In *Advances in neural information processing systems*, 2008.