Reaction Wheel Roll Stabilization and Control for an L1 High-Power Rocket

Sebastien Martinez*, Ethan Koh*, Renato Dell'Osso*, Rachit Gupta*, Dante Midei*, Dhananjay Manikandan*, Lucas Alonso-Munoyerro Anton**
Georgia Institute of Technology, Atlanta, Georgia, 30332, United States

This paper outlines the architecture and development of *Syndrome*, an L1 high-power rocket developed by the Guidance, Navigation, and Controls (GNC) project team of the Ramblin' Rocket Club (RRC) at Georgia Tech, designed to demonstrate closed-loop roll-rate stabilization using an internally mounted reaction wheel. The vehicle uses a reaction wheel to generate airframe reaction torque by accelerating a flywheel about the longitudinal axis, enabling roll control without external aerodynamic control surfaces. A discrete-time PID controller executes at 100 Hz and commands reaction-wheel torque using IMU gyroscope roll-rate feedback, while enforcing actuator feasibility through a torque bound that accounts for bus-voltage headroom and wheel-speed back-EMF limits. The control and sensing system is implemented on a custom avionics computer using an STM32H7-series microcontroller and a PCB integrating an IMU, barometric altitude sensing, recovery pyrotechnic channels, and onboard flash/SD storage. System software uses FreeRTOS to schedule concurrent sensing, control, logging, and supervision tasks. The paper presents the roll-axis dynamics and actuator models, the control law and saturation handling used to maintain stability under limited torque authority, and the hardware and real-time software structure supporting flight execution and data capture. The resulting architecture is intended to support hardware parameter identification and simulation-based validation, followed by an initial flight campaign demonstrating roll stabilization and subsequent extensions toward commanded roll-rate profiles.

I. Introduction

Uncommanded roll is a common disturbance in high-power rockets, driven by small geometric asymmetries and transient motor effects. Excess roll can degrade sensing and complicate recovery logic, and it becomes a limiting factor for any future attitude-control capability.

Syndrome is an L1 high-power rocket developed to demonstrate active roll stabilization using an internally mounted reaction wheel. A reaction wheel produces control torque through angular-momentum exchange and therefore provides roll authority without relying on external aerodynamic control surfaces. The vehicle implements a 100 Hz discrete-time PID roll-rate controller using IMU gyro feedback, and enforces actuator feasibility through a torque limit derived from wheel speed and bus voltage.

This paper describes the modeling, control implementation, and avionics hardware that support that objective. Section II presents a single-axis roll dynamics model and a reaction-wheel actuator model that captures voltage- and current-limited torque authority. Section III describes the discrete-time control law and the gain-selection approach. Section IV summarizes the embedded implementation on a custom STM32H7-based flight computer, including the real-time task structure and the PCB-level interfaces required for sensing, actuation, and logging.

II. System and Modeling

A. System Overview

Figure 1 summarizes the roll-rate control loop used on *Syndrome*. The IMU gyroscope measures the body roll rate at 100 Hz, and the controller compares this measurement to the desired roll rate (zero for stabilization) to form the error input to the PID controller.

The PID output is a commanded body torque about the roll axis. Because the reaction wheel cannot produce arbitrary torque at all wheel speeds and battery voltages, the controller is also provided an allowable torque limit at each update. This limit is computed from ESC telemetry, using the measured bus voltage and wheel speed to estimate how much motor current can be driven after accounting for back-EMF (Section II.C). The controller clamps its command to this limit, and the actuator model independently enforces the same physical limits in simulation.

*Student Member Undergraduate Category, **Student Member Graduate Category
AIAA numbers in order of names: *1810520, 1760396, 1924658, 1928813, 1930965, 1930905, 1928580*

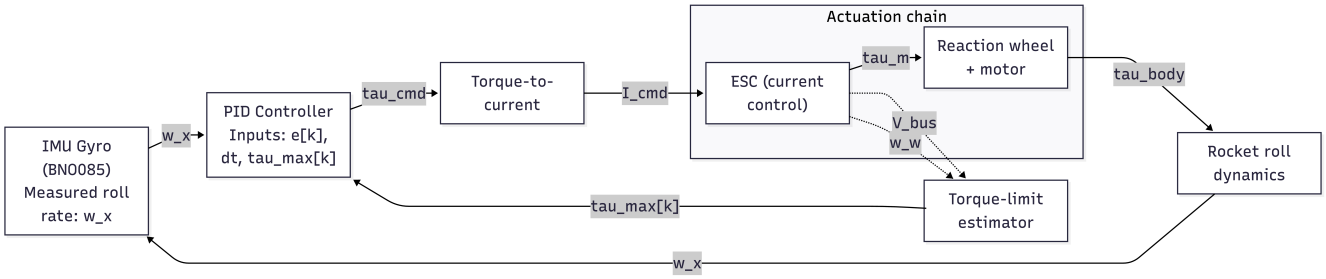


Fig. 1 System Overview

The flight computer sends commands to the ESC as a motor current request rather than as torque directly. The ESC regulates motor current internally, producing motor torque that accelerates the reaction wheel, and the equal-and-opposite reaction torque acts on the airframe to change the roll rate, closing the feedback loop.

B. Roll Dynamics Model

The rocket's roll dynamics are modeled as a single-axis rigid-body rotational system about the longitudinal body axis (x). Let ω_x denote the body-frame roll rate. The governing equation is

$$I_{xx}\,\dot{\omega}_x \;=\; \tau_x \tag{1}$$

where I_{xx} is the vehicle moment of inertia about the body x-axis and τ_x is the net applied roll torque on the airframe. We model the net torque as the sum of reaction-wheel actuation, aerodynamic roll damping, and lumped disturbances:

$$\tau_x \;=\; \tau_{body} \;-\; c_d\,\omega_x \;+\; \tau_{dist} \tag{2}$$

where τ_{body} is the reaction-wheel torque applied to the airframe, $c_d > 0$ is an effective roll-damping coefficient (primarily fin-generated), and τ_{dist} captures external disturbances and unmodeled effects. Substituting (2) into (1) gives the roll-axis plant used for controller design and simulation:

$$I_{xx}\,\dot{\omega}_x \;=\; \tau_{body} \;-\; c_d\,\omega_x \;+\; \tau_{dist}. \tag{3}$$

In simulation, the disturbance torque is modeled as a constant bias plus a single step change,

$$\tau_{dist}(t) \;=\; \tau_{bias} \;+\; \tau_{step}\,\mathbf{1}[t \geq t_0]\,, \tag{4}$$

where τ_{bias} represents a persistent roll-moment offset (e.g., manufacturing asymmetry or steady fin cant in the presence of airflow), and the step term captures an abrupt transition to a new disturbance level (e.g., launch-rail exit effects, thrust/motor transients, or a rapid change in wind-induced moment). This simple disturbance equation forces the controller to reject both steady disturbances and event-driven changes. For roll-rate stabilization with limited actuator authority, these two disturbance components are typically the dominant contributors to sustained error and saturation behavior, making (4) a sufficient stress model for gain tuning.

C. Reaction Wheel Actuator Model

The reaction wheel spin axis is aligned with the rocket body roll axis (x). We define τ_{body} as the torque applied to the airframe about $+x$, and τ_m as the motor-produced torque on the wheel rotor about $+x$. By Newton's third law, the torque transmitted through the motor produces an equal and opposite reaction torque on the airframe, so

$$\tau_{body} \equiv -\,\tau_m \tag{5}$$

Let ω_w denote the reaction wheel angular speed about $+x$. The wheel speed evolves according to the wheel rotational equation of motion

$$I_w\dot{\omega}_w \;=\; \tau_m(I,\omega_w) \;-\; \tau_f(\omega_w), \tag{6}$$

where I_w is the reaction wheel rotational inertia. The resisting torque $\tau_f(\omega_w)$ models parasitic losses in the wheel assembly. We use a viscous-plus-Coulomb model,

$$\tau_f(\omega_w) \equiv b\,\omega_w \;+\; \tau_c\,\mathrm{sgn}(\omega_w), \tag{7}$$

where the viscous term $b\,\omega_w$ captures approximately speed-proportional losses (e.g., bearing shear losses and other drag mechanisms that grow with speed), and the Coulomb term $\tau_c\,\mathrm{sgn}(\omega_w)$ captures approximately constant-magnitude friction

Table 1 Vehicle properties and uncertainties

Category	Property	Nominal	Uncertainty
Mass & Inertia	Longitudinal Moment of Inertia	0.502 kg m^2	±0.005 kg m^2
	Rotational Moment of Inertia	0.0049 kg m^2	±0.002 kg m^2
	Wet Mass	3392 g	±100 g
	Dry Mass	2640 g	±100 g
Geometry	Rocket Length	64.751 in	±0.2 in
	Body Tube Outer Diameter	4.014 in	±0.04 in
	Center of Gravity	43.364 in	±2.14 in
	Center of Pressure	48.307 in	±3.701 in
	Fin root chord	8 in	±0.01 in
	Tip root chord	2.8 in	±0.01 in
	Fin height	4.5 in	±0.01 in
	Fin sweep angle	41.6°	±0.05°
Flight Performance	Stability	2.17 cal	±0.04 cal
	Thrust-to-Weight Ratio at launch	4.71	–
	Estimated Apogee	2866 ft	±300 ft
System	Target α compensation of system	180 deg/s^2 (3.14 rad/s^2)	–
	Nominal Bus Voltage	18.5 V	–
	Maximum Current Supported by ESC	50 A	–

that opposes motion (e.g., bearing contact friction). In hardware, b and τ_c will be identified via coast-down or steady-state torque–speed tests.

For a current-controlled ESC operated in torque mode, we approximate the motor-produced torque as proportional to commanded motor current,

$$\tau_m(I, \omega_w) \;\approx\; K_t\, I, \tag{8}$$

where K_t is the motor torque constant. While (8) implies torque is proportional to current, the achievable current (and thus torque) is limited by both a hardware current ceiling $I_{\max}$ and available bus voltage headroom against motor back-EMF. Bus voltage balance can be approximated using an equation provided by Tyler Veness [1]. While this equation is discussed in the context of a brushed DC motor, brushless DC motors, such as on *Syndrome*, abide by the same relationship. Using this standard voltage balance approximation

$$V_{\text{bus}} \;\approx\; K_e\, \omega_w \;+\; I\, R, \tag{9}$$

where K_e is the motor back-EMF constant and R is the effective winding resistance, the speed-dependent current availability is modeled as

$$I_{\text{avail}}(\omega_w, V_{\text{bus}}) \equiv \frac{\max(0,\; V_{\text{bus}} - K_e \omega_w)}{R}. \tag{10}$$

The actuator simulation enforces the resulting torque bound

$$\tau_{\max}(\omega_w, V_{\text{bus}}) \equiv K_t\, \min(I_{\max},\; I_{\text{avail}}(\omega_w, V_{\text{bus}}))\,, \tag{11}$$

and saturates the requested torque accordingly. K_t and R are taken directly from the motor datasheet; K_e can be derived from K_t via standard motor constant conversions, provided consistent units are used.

D. Vehicle Properties and Uncertainties

The rocket was designed on OpenRocket and various parameters including wet mass, dry mass, length, etc., were directly obtained as a result from the software. Other parameters including Thrust-To-Weight Ratio at launch and estimated apogee, were obtained from an OpenRocket Simulation with launch conditions being set at International standard atmosphere and a minimal wind speed of 2m/s.

Tolerances were derived from a variety of methods. Through manual testing on body tubes and the implied 1 mm tolerance on the retailer page, we were able to create a conservative estimate for the diameter of the rocket. For measurements like rocket length, center of gravity, and fin parameters, the tolerance is defined by our manufacturing capability and the time/resources we are able to contribute.

III. Control Design and Tuning

A. PID Architecture

Syndrome regulates body-frame roll rate about the longitudinal axis (x) using a discrete-time PID controller executed at 100 Hz. Let $k \in \mathbb{Z}_{\geq 0}$ index the controller update step, with nominal timestep $dt = 0.01$ s. To reflect real-time scheduling effects in flight software, we model a variable effective update interval dt_k about this nominal value, capturing occasional late execution due to missed scheduling quanta and small additional execution latency. The PID computation at each step takes as inputs the roll-rate error $e[k]$ and an allowable torque bound $\tau_{\max}[k] > 0$, and outputs a desired body torque command $\tau_{\text{cmd}}[k]$ about $+x$.

The error signal is defined as

$$e[k] \;=\; \omega_{des}[k] - \omega_{meas}[k], \tag{12}$$

where $\omega_{meas}[k]$ is the measured roll rate from the IMU gyro and $\omega_{des}[k]$ is the *desired* roll rate. For strict roll stabilization we set $\omega_{des}[k] \equiv 0$, although $\omega_{des}[k]$ can be replaced with a time-varying roll-rate profile if a commanded roll maneuver is desired. The controller operates on the most recently available gyro sample, so $\omega_{meas}[k]$ incorporates sensor bias/noise and a measurement latency consistent with finite-rate sampling.

To prevent integrator dominance when actuator authority is limited, we budget a fixed fraction $\gamma \in (0, 1)$ of the available torque bound to the integral contribution. In this work we use $\gamma = 0.5$ (configurable), reserving roughly half of the available torque authority for the proportional and derivative terms during transients. Define

$$\tau_{I,\max}[k] \;\equiv\; \gamma\, \tau_{\max}[k], \qquad x_{I,\max}[k] \;\equiv\; \frac{\tau_{I,\max}[k]}{K_i}. \tag{13}$$

Because $\tau_{\max}[k]$ may vary with actuator state, the permitted integral budget can shrink between control steps. We therefore pre-clamp the stored integral state from the previous step to the current bounds:

$$x_I^0[k] \;=\; \text{clamp}\big(x_I[k-1],\; -x_{I,\max}[k],\; x_{I,\max}[k]\big)\,, \tag{14}$$

where $\text{clamp}(\cdot)$ denotes saturation to the provided lower and upper bounds.

We compute the (pre-clamp) torque command in position form, similar to Kiam Heong Ang et al.'s three-term PID formula [2]:

$$\tau_{\text{pre}}[k] \;=\; K_p\, e[k] \;+\; K_i\, x_I^0[k] \;+\; K_d\, \dot{e}_f[k], \tag{15}$$

where $\dot{e}_f[k]$ is a filtered discrete derivative of the error. The derivative term is computed from a low-pass filtered discrete derivative to reduce amplification of gyro noise:

$$\dot{e}_{raw}[k] \;=\; \frac{e[k] - e[k-1]}{dt_k}, \qquad \dot{e}_f[k] \;=\; \beta\, \dot{e}_f[k-1] + (1-\beta)\, \dot{e}_{raw}[k], \tag{16}$$

with $\beta = 0.8$ in this work. Although dt_k is variable, we keep β constant for simplicity. This makes the derivative filter's effective cutoff vary slightly from step to step; however, because the timing jitter is small relative to the nominal update period, this variation is small and has negligible impact on the tuned gains.

The integral state is updated as

$$x_I^\star[k] \;=\; x_I^0[k] + e[k]\; dt_k, \tag{17}$$

and the candidate updated state is clamped to the same bounds:

$$x_I^{\text{clamp}}[k] \;=\; \text{clamp}\big(x_I^\star[k],\; -x_{I,\max}[k],\; x_{I,\max}[k]\big)\,. \tag{18}$$

Because the reaction wheel has finite torque authority based on actuator state (Section II.C), the commanded torque is clamped using the provided bound $\tau_{\max}[k]$:

$$\tau_{\text{cmd}}[k] \;=\; \text{clamp}\big(\tau_{\text{pre}}[k],\; -\tau_{\max}[k],\; \tau_{\max}[k]\big)\,. \tag{19}$$

We apply conditional-integration anti-windup: if the controller is clamped and the error would drive $\tau_{\text{pre}}[k]$ further into the clamp, we freeze the integrator at its step-initial feasible value; otherwise we accept the updated clamped value:

$$x_I[k] \;=\; \begin{cases} x_I^0[k], & \text{if } \big(\tau_{\text{pre}}[k] > \tau_{\max}[k] \text{ and } e[k] > 0\big) \text{ or } \big(\tau_{\text{pre}}[k] < -\tau_{\max}[k] \text{ and } e[k] < 0\big), \\ x_I^{\text{clamp}}[k], & \text{otherwise.} \end{cases} \tag{20}$$

Anti-windup is necessary because during actuator saturation, additional integral accumulation does not increase the applied torque; it instead stores excess integral action that manifests as overshoot and prolonged recovery once the actuator returns to the linear regime. Pre-clamping via (14) ensures the integrator remains within the current allowable budget when $\tau_{\max}[k]$ changes.

B. Tuning Problem Formulation

PID gains are tuned via simulation-based optimization over the parameter vector

$$\theta \equiv [K_p,\ K_i,\ K_d], \tag{21}$$

with $(K_p, K_i, K_d) \in \mathbb{R}_{\geq 0}$ and bounded search ranges chosen to ensure numerical stability and to constrain the optimizer to physically plausible closed-loop responses. Closed-loop performance for a given θ is evaluated in time-domain simulation using the roll dynamics model (Section II.B), the reaction wheel actuator and torque-limit model (Section II.C), and the discrete-time controller executed with nominal timestep $dt = 0.01$ s and variable effective update interval dt_k (Section III.A).

For a single simulation trial under a particular scenario draw s, we compute a per-trial cost $\ell(\theta; s)$ from the resulting time histories. The scalar objective value returned to the optimizer is then defined as a Monte Carlo (MC) aggregate over N independent trials,

$$J(\theta) \equiv \mathrm{Agg}\Big(\{\ell(\theta; s_i)\}_{i=1}^{N}\Big), \tag{22}$$

where $\mathrm{Agg}(\cdot)$ denotes an aggregation operator such as the sample mean augmented with a high-percentile penalty and an explicit penalty for failed trials.

Each trial samples a scenario s consisting of (i) initial conditions, (ii) external roll disturbances (Section II.B), and (iii) uncertain plant, actuator, sensor, and timing parameters. The current MC implementation varies the following quantities: initial roll rate $\omega_x(0)$; the roll-axis inertia I_{xx} and roll-damping coefficient c_d; reaction-wheel parameters I_w and friction coefficient b; actuator limits and electrical conditions through $I_{\max}$ and nominal V_{bus}; gyro imperfections through an additive bias, noise level, and a measurement latency; and control-task timing through discrete tick-sized delays (e.g., occasional +1 tick or +2 tick slips) plus a small additional bounded latency. Physical-property variations are drawn using the nominal values and uncertainties summarized in Table 1; other variations (ESC/electrical, sensor, and timing) are set to reflect the expected spread for the intended simulation studies.

This list is a proposed baseline set of MC variables. It can be extended to include additional uncertainty sources if higher-fidelity variation is needed, but the parameters above capture the dominant effects for roll-rate stabilization: changes in inertia and aerodynamic damping, changes in available torque authority via voltage/current limits, sensor bias/noise/latency, and small timing jitter that perturbs the discrete-time update.

Because each objective evaluation computes $J(\theta)$ from freshly sampled scenarios $\{s_i\}_{i=1}^{N}$, the mapping $\theta \mapsto J(\theta)$ is stochastic. Optimizing the MC aggregate encourages gains that perform well across variability rather than gains that overfit a single nominal trajectory.

C. Objective Function

For a single Monte Carlo trial under scenario draw s, we compute a scalar per-trial cost $\ell(\theta; s)$ from the closed-loop time histories over $t \in [0, T]$. The roll-rate error $e(t)$ is defined as in Section III.A. The per-trial cost is constructed from three averaged quadratic metrics:

$$\ell_e(\theta; s) \equiv \frac{1}{T}\int_0^T e^2(t)\, dt, \tag{23}$$

$$\ell_u(\theta; s) \equiv \frac{1}{T}\int_0^T \tau_{\mathrm{cmd}}^2(t)\, dt, \tag{24}$$

$$\ell_r(\theta; s) \equiv \frac{1}{T}\int_0^T \dot{\tau}_{\mathrm{cmd}}^2(t)\, dt, \tag{25}$$

where $\tau_{\mathrm{cmd}}(t)$ is the commanded body torque output by the PID controller after clamping to $\pm\tau_{\max}(t)$, and $\dot{\tau}_{\mathrm{cmd}}(t)$ is the time derivative of the commanded torque. In discrete-time implementation, the integrals are evaluated via trapezoidal integration over the simulation time vector (allowing nonuniform dt_k), and $\dot{\tau}_{\mathrm{cmd}}$ is approximated by a finite difference using the corresponding time spacing.

The per-trial cost is the weighted sum

$$\ell(\theta; s) \equiv w_e\, \ell_e(\theta; s) + w_u\, \ell_u(\theta; s) + w_r\, \ell_r(\theta; s). \tag{26}$$

The three components have different physical units: ℓ_e has units of $(\mathrm{rad/s})^2$, ℓ_u has units of $(\mathrm{N \cdot m})^2$, and ℓ_r has units of $(\mathrm{N \cdot m/s})^2$. The weights therefore serve two purposes: they compensate for the disparate magnitudes induced by unit differences, and they encode the intended semantic trade between roll-rate regulation, torque usage, and torque smoothness. In this work we use $(w_e, w_u, w_r) = (1.0,\ 10^{-4},\ 10^{-6})$, chosen to prioritize roll-rate regulation while weakly discouraging excessive torque usage and high-frequency torque activity.

For a given gain set θ, the optimizer is provided an aggregated objective value computed from N independent trials $\{s_i\}_{i=1}^{N}$. We use the sample mean augmented with a tail-risk penalty:

$$J(\theta) \;\equiv\; \frac{1}{N}\sum_{i=1}^{N} \ell(\theta; s_i) \;+\; \lambda\, \mathrm{P}_{95}\Big(\{\ell(\theta; s_i)\}_{i=1}^{N}\Big), \tag{27}$$

where $\mathrm{P}_{95}(\cdot)$ denotes the empirical 95th percentile across trials. We set $\lambda = 0.5$ to bias the optimizer toward gain sets that avoid poor outlier performance while still primarily optimizing mean behavior. The parameters (w_e, w_u, w_r) and λ are configurable and may be refined as the simulation and hardware characterization mature.

D. Optimization Method and Tuning Results

PID gains were selected using Bayesian optimization (BO), treating the objective $J(\theta)$ from Section III.C as a black-box function of $\theta \equiv [K_p, K_i, K_d]$. BO is a good fit because the decision space is low-dimensional, analytic gradients are unavailable, and each evaluation requires running a full closed-loop simulation.

The optimizer searches in log-space to span multiple orders of magnitude while maintaining numerical stability. Specifically, it searches over $(\log_{10} K_p, \log_{10} K_i, \log_{10} K_d)$ with bounds

$$\log_{10} K_p \in [-3, 3], \qquad \log_{10} K_i \in [-6, 2], \qquad \log_{10} K_d \in [-6, 2].$$

These ranges were chosen to (i) exclude trivially small gains that produce negligible control authority, (ii) avoid excessively large gains that cause numerical instability or persistent saturation against $\tau_{\max}$, and (iii) reflect typical relative scaling where K_i and K_d are often smaller than K_p for a 100 Hz discrete-time implementation.

Because $J(\theta)$ includes variability from disturbances, sensor imperfections, and timing jitter, each BO evaluation aggregates performance over a small batch of randomized trials rather than a single nominal run. This reduces the chance of selecting gains that perform well only for one specific scenario draw. New candidate gains are chosen using an "expected improvement for noisy objectives" acquisition rule that balances exploring uncertain regions against refining regions predicted to be promising.

BO is terminated when the best-so-far objective shows negligible improvement over a fixed patience window after a minimum number of evaluations, or when a safety cap is reached. The best-performing gain set under this procedure is used as the controller gains for *Syndrome*'s flight software. Detailed robustness sweeps and additional validation are deferred to future work.

IV. Avionics and Hardware Implementation

A. Real-Time Architecture

Syndrome's flight software architecture was motivated by integration and timing issues observed in the GNC project team's prior *Scarlet* vehicle, a gimbaled thrust-vector-control (TVC) rocket developed during the 2024–2025 academic year. In *Scarlet*, control, sensing, recovery, and logging were organized as a single event-driven loop. The rocket had an effective control-loop frequency on the order of ~ 25 Hz, and was limited by blocking SD-card writes and blocking barometer transactions in the main loop. These design constraints motivated a real-time scheduling approach for *Syndrome*.

Syndrome uses FreeRTOS, an open-source real-time operating system (RTOS). An RTOS organizes flight software into separate *tasks* so that time-critical functions can run at predictable rates [3]. Each task is a specific piece of flight code (e.g., read the IMU, run the controller) that can run periodically (at a specified *rate*) or in response to an event, and each task has a set *priority*. FreeRTOS uses preemptive priority scheduling: at every *tick* (normally 1000 ticks per second), it runs the highest-priority *ready* task, and a newly-ready higher-priority task can interrupt a lower-priority one.

Syndrome's primary timing requirement is a nominal 100 Hz roll-rate control update. The control task is released every 10 ms and meaningful preemption of the control task is primarily due to the pyro event; otherwise, jitter is dominated by tick granularity and small runtime variation. Rather than assuming a fixed timestep, the flight software measures the true elapsed time between successive control-task start timestamps and uses this value as dt_k (Section III.A), similar to the simulator.

Sensor acquisition is handled by separate periodic tasks that publish the most recent measurements to a shared `RocketState` structure, and the control task consumes the latest available values each cycle. `RocketState` is the shared "current snapshot" of the vehicle and contains the latest sensor measurements needed by control and event detection, the current flight state, the control timing metadata, and a fault/status bitmask that is logged for post-flight diagnosis.

Unlike *Scarlet*'s single-loop structure, where a long or blocking operation could stall the entire system and reduce the control update rate, *Syndrome* isolates deployment into its own pyro task so that firing does not block control or sensor updates. The pyro task is event-driven: when apogee is detected, it briefly runs at high priority to arm and start the firing sequence, then a hardware timer/driver turns the channels off after the commanded duration without requiring the pyro task (or CPU) to remain occupied for the full one-second window.

Table 2 FreeRTOS task set with nominal rates and priorities

Task	Rate (Hz)	Priority	Purpose
Pyro	event-driven	0/5	Initiates pyro firing at apogee (priority 5 only during the trigger sequence)
Control	100	4	Computes τ_{cmd}; measures dt_k; computes $\tau_{\max}$ and applies anti-windup
IMU	100	3	Publishes gyro roll rate ω_x and attitude estimates
Barometer	50	2	Publishes pressure/altitude for event detection and logging
Supervisor	10	2	Monitors task heartbeats; transitions to safe-mode on missed deadlines
Storage	50	1	Buffers data to flash during flight; transfers to SD post-flight

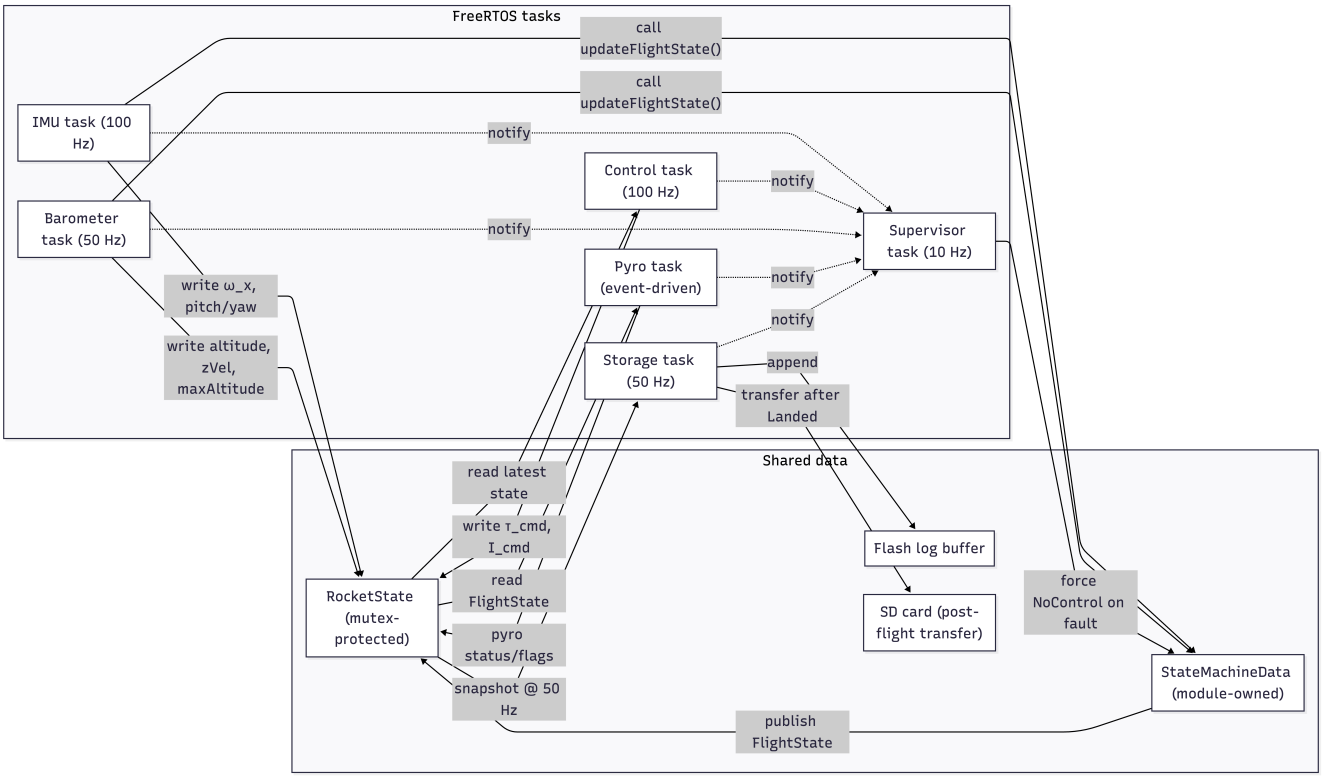


Fig. 2 Task Interactions

FreeRTOS' preemptive scheduling introduces concurrency hazards because multiple tasks share data structures. *Syndrome* mitigates this with mutual exclusion locks (mutexes). Sensor tasks acquire the `RocketState` mutex, update a coherent set of fields for one sample, and release it. The control task briefly acquires the same mutex to copy `RocketState` into a local snapshot, then releases it and performs the control computation without holding locks. This prevents mixed-time reads while keeping mutex hold times short, which bounds added jitter in the periodic loop.

Flight phase is tracked by an event-driven state machine (Figure 3) rather than by a dedicated periodic "state task." Transition checks are invoked at the end of relevant update tasks (e.g., IMU updates for launch detection and barometer updates for apogee/landing detection), so transitions occur promptly after new data arrives. State-transition bookkeeping is stored separately in a `StateMachineData` structure owned by the state machine module and protected by its own mutex.

Data logging is structured to avoid the blocking SD-card write behavior that degraded control-loop frequency in *Scarlet*. During flight, the storage task writes `RocketState` snapshots to an onboard flash device at 50 Hz. Based on a comparison of flash chip data from Peter Desnoyers [4] and SD card data from Robbert Krawinkel [5], flash chips have significantly lower latency than SD cards and so block for a much shorter duration. After landing is detected, the system transfers the flash log to an SD card using a filesystem layer (FatFS). This two-stage approach keeps the SD card off the critical path during flight while preserving full-rate logs for post-flight analysis.

Finally, a supervisor task provides fault containment. Each supervised task periodically emits a heartbeat notification. The supervisor checks whether expected heartbeats have been observed within a timeout window to detect stalled execution. If a critical task (notably IMU or control) stops reporting, the supervisor transitions the system to `NoControl`, disabling control outputs while allowing recovery logic and data preservation to continue. The supervisor also records faults into the `RocketState` fault bitmask so that timing and health issues are visible in the flight log.

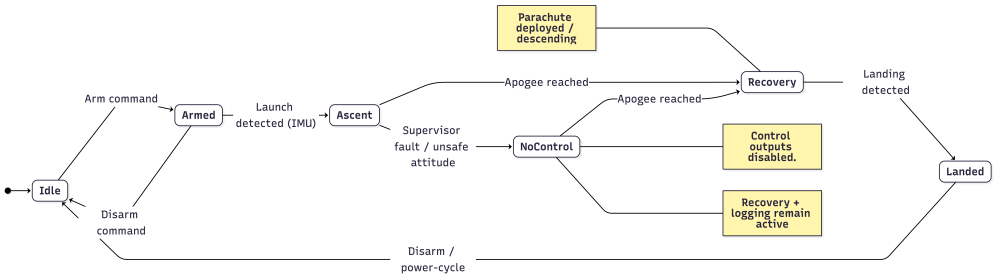


Fig. 3 Flight State Machine

B. Control Loop Implementation

The roll-rate control update executes inside the 100 Hz control task. At the beginning of each cycle, the task reads a high-resolution timestamp from a hardware timebase driven by an external oscillator. This is more precise than using RTOS tick counts alone, which are quantized to the tick period and cannot represent sub-tick jitter. The control task stores its most recent start timestamp and computes the measured update interval

$$dt_k \;=\; t_k - t_{k-1},$$

which is passed into the controller equations (Section III.A). This measured dt_k is also the timing value referenced by the simulation's jitter model: jitter is treated as discrete tick-sized delays with small additional latency, matching how a tick-based scheduler behaves in practice.

The control task consumes a coherent snapshot of `RocketState` (copied under mutex) and uses the most recent gyro roll rate ω_x as the feedback measurement. It then determines the allowable torque bound $\tau_{\max}[k]$ from ESC telemetry and actuator state, and finally computes the commanded torque $\tau_{\text{cmd}}[k]$ with anti-windup and saturation (Section III.A).

The control task also owns the ESC UART link. ESC telemetry (bus voltage V_{bus} and wheel speed ω_w) and current commands share the same UART bus, so splitting telemetry into a separate task would require explicit bus arbitration. Instead, the control task performs the ESC UART transactions at the start of each control cycle: it requests V_{bus} and ω_w, computes $\tau_{\max}[k]$ using Section II.C, then computes and transmits the control command. This keeps telemetry and torque-limiting aligned with the control update while avoiding shared-bus concurrency hazards.

The flight computer does not command torque directly; it converts the desired body torque $\tau_{\text{cmd}}[k]$ into a requested motor current using the motor torque constant and the sign convention $\tau_{\text{body}} = -\tau_m$:

$$I_{m,\text{cmd}}[k] \;=\; \frac{\tau_{m,\text{cmd}}[k]}{K_t} \;=\; \frac{-\tau_{\text{cmd}}[k]}{K_t},$$

and transmits $I_{m,\text{cmd}}[k]$ to the ESC over UART. The ESC performs internal current regulation to realize the requested motor torque, subject to its current ceiling and available voltage headroom.

If ESC telemetry is temporarily unavailable, the control task does not block waiting for it; instead, it falls back to a conservative torque bound for that cycle to preserve the 100 Hz deadline. Concretely, the fallback bound is chosen as a fraction of the previous cycle's permitted torque:

$$\tau_{\max}[k] \leftarrow \alpha\, \tau_{\max}[k-1], \qquad \alpha = 0.9,$$

which intentionally reduces authority until telemetry returns. To support this, `RocketState` stores the previous-cycle $\tau_{\max}$ used by control. This policy prevents the controller from assuming unrealistically high available torque when actuator state is uncertain, while keeping the control loop running on-time.

C. Platform-Agnostic Driver Architecture

Syndrome's flight software is structured so that device drivers are decoupled from the low-level communication mechanism used to exchange bytes with each peripheral, thus improving portability and testability. A driver is the portion of the codebase that implements device-specific behavior, such as configuring an IMU output data rate or reading a barometer pressure register.

The architecture is organized into three layers. At the top, the *driver layer* defines device-centric operations (e.g. reading altitude from barometer) and translates them into register reads/writes or protocol transactions. At the bottom, the *hardware abstraction layer* (HAL) provides microcontroller-specific access to peripherals such as SPI, I^2C, UART, timers, and GPIO. Between driver and HAL, the *transport layer* exposes a narrow, device-agnostic interface for performing byte transfers. Drivers depend only on this transport interface and therefore remain agnostic to the underlying bus and microcontroller.

This decoupling is implemented using C++ *templates*, a language feature that allows a class or function to be parameterized by a type. Each driver is written as a templated class that accepts a *transport type* as a template parameter. The driver can then call transport operations (e.g., `read` and `write`) without committing to a specific bus implementation. Because templates are instantiated at compile time, the resulting code is specialized for the chosen transport and does not require run-time dispatch, as noted by Scott Haney and James Crotinger [6].

To ensure that only valid transport types are used to instantiate the templated drivers, *Syndrome* employs *concepts*, a feature introduced in C++20. A concept is a compile-time contract that specifies what operations a template parameter must provide, including function signatures and associated types. Each driver constrains its transport parameter with a corresponding transport concept, so a broad class of integration mistakes are caught at compile-time, not run-time.

Compile-time templating is preferred to run-time polymorphism with virtual base classes, especially in real-time systems. Run-time polymorphism introduces an additional indirection through a virtual function table (vtable) [7], which can add overhead and timing variability in tight periodic loops. In contrast, template-based composition produces direct calls that the compiler can inline and optimize, and interface correctness is verified at compile time.

Finally, this structure enables off-target testing and simulation by substituting a mock transport implementation in place of the hardware transport. The mock transport can return scripted register values, inject communication errors, or emulate timing behavior, allowing driver logic to be validated without requiring the target microcontroller or physical hardware. This same mechanism supports simulation integration by allowing drivers to be exercised against synthetic sensor data streams while preserving the same driver interfaces used in flight.

D. Hardware Interfaces and Electrical Constraints

The avionics PCB is designed to satisfy three system requirements: (i) sustain a 100 Hz control update while also sampling sensors and logging data, (ii) keep the IMU measurements reliable in the presence of electrical noise created by high-current devices and power conversion on the board, and (iii) provide robust electrical interfaces for the reaction wheel actuator, recovery system, and post-flight data retrieval.

Compute and I/O are centered on an STM32H723ZG (ARM Cortex-M7). The MCU was selected because its processing capability and peripheral set support the full FreeRTOS task set used in flight, including periodic sensing and control alongside lower-priority logging and supervision, without reducing the 100 Hz control rate. The device also provides sufficient independent serial interfaces to keep critical links dedicated rather than shared. In particular, the reaction wheel ESC is connected via a dedicated UART on a 7-pin JST connector. This avoids contention between control-critical motor commands and other peripheral traffic. The Maytech ESC runs VESC firmware independently, so the flight computer's role is limited to transmitting current commands and reading telemetry over UART.

Sensors were chosen to support both control feedback and flight-state estimation with low integration risk. The BNO085 IMU provides roll-rate feedback and attitude information, and its onboard fusion outputs quaternion attitude estimates. This reduces the amount of signal processing required on the MCU compared to implementing full attitude estimation from raw accelerometer and gyro registers. The BMP390 barometer provides altitude estimates used for event detection and logging, and was selected based on prior team experience and its proven reliability in previous launches. Both sensors are integrated as breakout modules, which simplifies rework and replacement during development.

For development and troubleshooting, the PCB includes a JTAG header for programming and inspection, a hardware reset switch, and distributed test points to support oscilloscope and multimeter measurements during bring-up. A piezo buzzer provides a simple state indication (e.g., idle/armed/landed) when no wired connection is available.

The PCB layout minimizes noise coupling into the MCU and sensors. Some parts of the system draw large currents or create fast voltage transitions, which can inject noise into nearby signal traces and power rails. Examples include the DC–DC buck converters (which generate 3.3 V and 5 V from the battery by rapidly switching current), the pyrotechnic firing channels (which draw a short, high-current pulse when igniting an e-match), and the wiring associated with the motor/ESC (which carries high currents during wheel spin-up and torque changes). To prevent this noise from corrupting sensor data or causing resets, high-power components and high-current paths are placed away from the MCU and sensors. The MCU is placed centrally and the IMU and barometer are located nearby to minimize trace length and loop area. Continuous ground planes provide a low-impedance return path and help shield sensitive traces. Local decoupling capacitors at MCU power pins suppress supply ripple and fast transients that could otherwise disturb sensor sampling.

Power distribution is designed so that large load changes do not propagate into the control computer. The avionics bay contains three batteries. A dedicated 3S battery (11.1 V nominal) powers the flight computer through an XT-60 connector, and the PCB implements reverse-polarity and overcurrent protection on this input to reduce the risk of damage during integration. Two regulated rails are generated from the 3S input using NR111E buck converters. The 3.3 V rail powers the MCU and all logic-level devices, while the 5 V rail is reserved for the three pyrotechnic channels. Keeping pyro power on a separate regulated rail reduces the probability that firing transients disturb the MCU supply.

The reaction wheel actuator is powered separately by a 5S battery (18.5 V nominal) connected to the ESC. This separation prevents motor current changes from pulling down the flight-computer supply or injecting noise into the sensor and MCU rails. It also defines the software's contract with hardware limits: the controller computes $\tau_{\max}$ using ESC telemetry and voltage

headroom (Section II.C), and if telemetry is unavailable it falls back to a conservative bound for that cycle (Section IV.B) to preserve real-time execution and avoid commanding torque that the electrical system may not be able to deliver. Finally, a 9 V battery powers a backup commercial flight computer dedicated to recovery initiation, providing an independent deployment path if the custom avionics experiences a fault.

E. Hardware Design

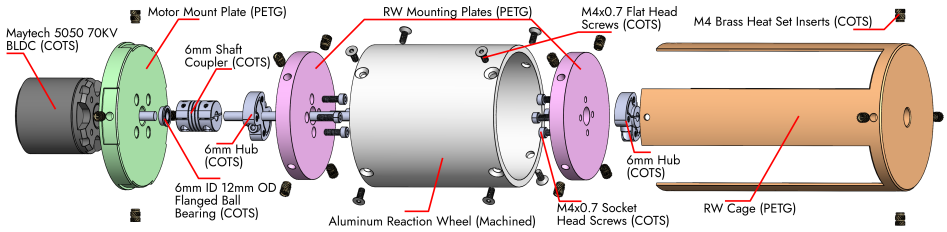
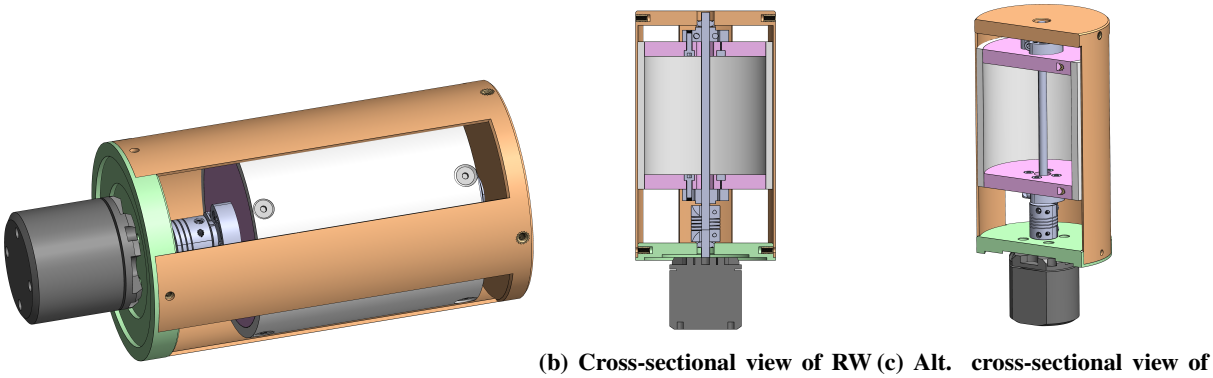


Fig. 4 Exploded view of reaction wheel mount assembly



(a) Reaction wheel mount assembly

(b) Cross-sectional view of RW mount

(c) Alt. cross-sectional view of RW mount

Fig. 5 Reaction Wheel Views

1. Reaction Wheel Mount Design

Design The reaction wheel was designed to make use of as many COTS and 3D printed parts as possible to enable rapid iteration. A full list of components is shown in 4. Brass heat-set inserts were used when screwing into 3D printed parts. The mount is secured to the airframe via four screws that pass through the lower body tube and the outer cage structure, anchoring into the heat-set inserts of the main mounting chassis. An additional four screws secure the upper section of the mount to the rocket. As shown in 5b, a 6mm shaft is connected via a coupler to a 150mm extension shaft. A mounting hub secures this assembly to a 3D-printed adapter, which supports the aluminum reaction wheel from either end.

Brushless Motor Selection The Maytech 5055 70 KV Brushless DC (BLDC) Motor was selected for this system. It is the smallest BLDC motor available that has an integrated Hall-effect encoder, which is essential for Field Oriented Control (FOC) of the motor via the Electronic Speed Controller (ESC). To maximize the torque output, the 70KV variant was paired with a 5S battery (18.5V nominal voltage) with 1550mAh capacity and a 100C maximum continuous discharge rating. The maximum speed of the motor powered by an 18.5V battery is 1295.00 RPM as calculated using equation 28. The maximum torque of the motor is 4.093 N·m as calculated using equation 29.

$$RPM_{max} = K_V \cdot \text{Battery Voltage} \tag{28}$$

$$\tau_{max} = K_t \cdot \text{Battery C Rating} * \text{Battery Capacity} \tag{29}$$

Reaction Wheel Sizing The reaction wheel was sized with the goal of counteracting an angular acceleration of 3.14 rad/s^2 = 180 deg/s^2 from *Syndrome's* launch until apogee (t=15s) into the flight. It was unknown if the torque commands to the BLDC motor, even with the more accurate field-oriented control (FOC), would remain accurate around the no-speed (0%-10%) regime of the BLDC, and thus, the reaction wheel was sized around an assumption that it would begin spinning during launch at 55%

of its maximum speed and vary between 10% and 100% of its maximum speed to counteract the applied angular accelerations. Future testing with hardware can confirm if this (0%-10%) assumption is true and the mass of the reaction wheel system can be adjusted accordingly. The reaction wheel mass is 357.347g, which was roughly 15% higher than the calculated 310.449g necessary for the target angular acceleration.

2. Overall Rocket Design

Syndrome is a Level 1 single-deployment rocket constructed using standard high-power rocketry techniques. OpenRocket v24.12 was used to create the initial rough design of the rocket and calculate basic flight parameters, many of which are listed in 1.

Due to the considerable mass introduced by the reaction wheel system and avionics, we chose Blue Tube 2.0, a more rigid body tube material which balanced durability with manufacturability. A phenolic coupler tube joins the upper and lower body tube sections together and is attached to the lower body tube section with 4 screws, as shown in 6. The avionics bay along with the black powder charge wells are housed within the coupler tube. The fins will be laser cut from 1/4" basswood sheets and interlock with the motor inner tube. Propulsion is provided by an HP-I65W solid rocket motor. This motor was selected for its ideal balance of high impulse of 635Ns with a long burn time of 7.9s. This maximizes the duration of powered flight under thrust for reaction wheel testing.

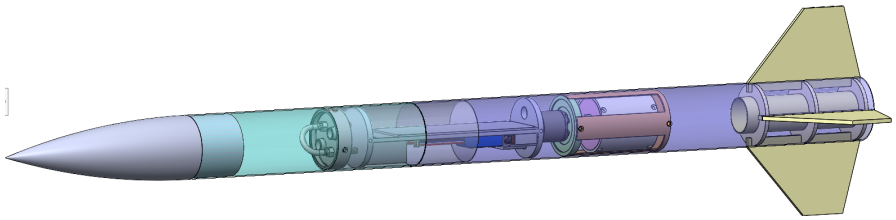


Fig. 6 Syndrome Full Rocket

V. Conclusion and Next Steps

This paper presented the roll-control architecture for *Syndrome*, an L1 high-power rocket designed to demonstrate reaction-wheel roll stabilization. The roll-axis rocket model, voltage- and current-limited actuator model, 100 Hz discrete-time PID controller with saturation handling, and FreeRTOS-based embedded implementation with measured update intervals were presented.

The next step is to replace assumed constants with measured values and use those measurements to validate and refine simulation before flight. Reaction-wheel parameters (b, τ_c, I_w) can be identified using coast-down and step-current tests while logging wheel speed. Motor and electrical constants (K_t, K_e, R) and the torque-limit relation $\tau_{\max}(\omega_w, V_{\text{bus}})$ can be validated by commanding known currents and measuring acceleration and terminal voltage across wheel speeds and bus voltages. Vehicle parameters (I_{xx} and the effective roll damping c_d) can be refined through inertia measurement and controlled spin-down experiments. Sensor and timing parameters can be characterized on the integrated flight computer by logging IMU bias/noise, effective measurement latency, and control-task timestamps to quantify the dt_k distribution.

With hardware-calibrated parameters, closed-loop simulations can then sweep the measured uncertainty ranges, stress voltage-sag and saturation cases, and retune gains if needed prior to flight.

References

[1] Veness, T., *Controls Engineering in the FIRST Robotics Competition*, 2017, Chap. 13.

[2] Kiam Heong Ang, Y. L., G. Chong, "PID control system analysis, design, and technology," *IEEE Transactions on Control Systems Technology*, 2005.

[3] Walter Cedeño, P. L., "An Overview of Real-Time Operating Systems," *SLAS Technology*, 2007.

[4] Desnoyers, P., "Empirical Evaluation of NAND Flash Memory Performance," *ACM SIGOPS Operating Systems Review*, 2010.

[5] Kräwinkel, R., "The effect of writing and transmitting SD card data on the consistency of SD card write performance," Thesis, University of Twente, Enschede, 2020.

[6] Scott Haney, J. C., "How templates enable high-performance scientific computing in C++," *Computing in Science Engineering*, 1999.

[7] Mengchi Zhang, T. R., Ahmad Alawneh, "Judging a Type by Its Pointer: Optimizing GPU Virtual Functions," *ASPLOS '21: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.