Above the Clouds: New Software Challenges in Space Computing

Abstract

Satellite-backed services have become an essential component of everyday life, in areas such as navigation, Internet connectivity and imaging. The collapsing cost of launching to space has disrupted the way satellites are deployed, shifting the industry from a model of few expensive fault-tolerant high-orbit satellites to arrays of commodity low-cost SmallSats in low-Earth orbit. However, satellite software hasn't kept up with the hardware trends, and missions are still using the ad-hoc flight software infrastructure built for expensive one-off missions in high-altitude orbits, wherein operators manually deploy software to each satellite individually. This approach is woefully inadequate in the new emerging SmallSat operational model, where an operator needs to manage hundreds of "wimpy" satellites with varying hardware capabilities under intermittent communication. Furthermore, SmallSat operators increasingly "rent out" their infrastructure to third parties, and need to support the workloads of multiple different tenants on the same satellites, which raises the classic problems of isolation and security similar to cloud computing, but in the much more constrained hardware environment of space. In this paper, we describe the new research questions introduced by this operational model. We also sketch the design of a novel lightweight eBPF-based runtime for fleets of multi-tenant, heterogeneous and intermittently-connected satellites.

CCS Concepts

• **Computer systems organization → Real-time systems**; • **Software and its engineering → Software creation and management**.

Keywords

satellite computing, flight software, bytecode languages

1 Introduction

Spacecraft have become a critical, though often invisible, component of everyday life. Beyond the reliance we all have on GPS, Spacecraft data has been valuable in many new fields, such as global food supply [1], web mapping [2] and increasingly real-time weather reports [3], which help governments prevent and rapidly react to natural disasters such as floods and forest fires. Similarly, satellite Internet services such as Starlink [4] and Hughes [5] provide important lifelines for remote and disaster-struck areas. Meanwhile, spacecraft such as the James Webb Space Telescope and the Mars Perseverance rover aid in scientific discovery.

The rapid rise in spacecraft deployment has been driven by an exponential decrease in launch costs, from $88.5K[1] per kilogram in 1981 on the Space Shuttle to just $1.4K on SpaceX's Falcon Heavy today [6]. The reduction in launch costs has resulted in operators moving away from few expensive, custom-built satellites running bespoke software stacks in favor of many cheap, commodity SmallSats running commodity OSes such as Linux [7]. This mirrors the shift from large expensive mainframes to clusters of commodity servers during the rise of cloud computing.

The introduction of off-the-shelf parts has opened up an emerging market for multi-tenant payloads, similar to how compute is allocated in public clouds [8]. In such an arrangement, clients can host their instruments onboard a vendor's spacecraft, saving on costs such as testing and certification [7]. Each satellite thus serves multiple different customers, all of whom will need to run local applications onboard the flight computer to interface with the instruments and preprocess data. Similar to isolation in the cloud, operators need to ensure isolation between each tenant's processes, while also allowing them access to low-level hardware interfaces.

The current cloud computing paradigm of isolation through virtualization cannot adequately address this challenge, as the overhead from virtual machines could cause the system to miss real-time requirements. This is further exacerbated by the less powerful chips built for spacecraft, some of which do not have hardware virtualization support or even a memory management unit [7]. While previous work in the IoT space have shown that a bytecode-based approach can be effective at providing multi-tenancy in low-power real-time applications [9], bytecode is far less performant than that native code, blocking its adoption for compute-heavy use cases such as machine learning or image recognition.

The space environment also introduces novel compute and communications challenges. Flight software must also ingest data from high-fidelity sensors that can emit gigabytes of data each second, while also taking into account realtime limitations on various processes [10]. While these are well-studied challenges in the cloud computing space, doing so with the low-power processors spacecraft use [2] due to their limited thermal and power headroom is far more challenging. Making matters worse, as more spacecraft are launched, bandwidth between satellites and ground stations is becoming a critical bottleneck [11]. This will require future spacecraft to be

[1] Normalized to 2024 dollars.

capable of complex onboard compute, while minimizing runtime and staying within the spacecraft's power envelope.

While current cloud computing paradigms assume that nodes exist in a datacenter where compute and bandwidth is plentiful and latency is low, these satellites are multi-tenant, weak, heterogeneous nodes that experience intermittent connectivity in low-bandwidth situations. Bringing cloud-like multi-tenant computing to such an environment comes with an array of technical challenges. To this end, we aim to highlight open research questions surrounding applying systems principles to this relatively new setting of managing vast arrays of "wimpy" satellites running multi-tenant workloads.

Furthermore, we provide a high-level design of a new satellite software runtime that enables operators to adapt to new situations without rewriting large portions of it, while enforcing isolation. To do so, a key element of our design revolves around compartmentalizing software features into extensions. We also show how these extensions should be written in an ISA-independent bytecode to support the heterogeneous low-bandwidth environment that can run in a sandbox environment. We find that the eBPF bytecode is a promising approach for our SmallSat use case. eBPF's existing kernel use case emphasizes memory safety and isolation, an extremely important consideration for satellites. eBPF's lightweight, cross-ISA nature also reaches the performance of native binaries while being more bandwidth-efficient than other bytecodes.

2 Extended Motivation

Software is critical to the correct operation of all modern spacecraft. While the earliest satellites merely had to send out radio pings at regular intervals, today's flight software has a far wider array of responsibilities, which requires significant computational power to both plan [12] and execute. For example, Earth observation satellites pre-process queries with lightweight machine learning models before downlinking [10, 13]. Similarly, to minimize latency, communication satellites often run packet processing tasks before forwarding network traffic to a ground station [11]. Deep space exploration spacecraft also use complex image processing methods to accurately determine their location [14].

Historically, satellite launches tended to be large, expensive, one-off missions with little to no tolerance for error. As there were relatively few active satellites at any one time, it was possible to control each satellite manually with bespoke software [15]. Even today, state-of-the-art satellite workflows require teams to manually write customized sequences, essentially a list of function calls, every day for every satellite [16]. After executing each workflow, the satellite sits idle until the next sequence is uploaded, which tends to occur once a day. Recent advances in LEO SmallSats have created four new challenges, which we outline in this section.

Intermittent connectivity. Due to their lower orbits, SmallSats also have a lower field-of-view of the Earth. Thus, more satellites are required in LEO constellations to provide the same coverage compared to those in higher orbits, further increasing the LEO SmallSat population. For example, high-orbit constellations such as those providing GPS need only 24 satellites to provide coverage to the entire world [17]. In contrast, Planet needs over 150 LEO SmallSats to provide full coverage of the Earth [2] while Starlink uses over 4,000 active SmallSats to serve its connectivity needs [18]. The current manual approach to developing spacecraft software [13] is inadequate for such a large number of satellites.

SmallSats also experience intermittent and sometimes limited connectivity with ground stations, since line-of-sight between the satellite and ground station is needed for satellite communication and data collection [19]. Traditional satellites are less affected by this limitation because they orbit at altitudes 160× higher than LEO, and are thus virtually guaranteed to be within line-of-sight to a ground station. However, the lower field-of-view of individual SmallSats means that ground station connections are much more intermittent. For example, a typical ground station will have line-of-sight to a SmallSat orbiting above it for only ~100 seconds each day. With traditional X-band radios operating at 4 Mbps [19], this means we are have ~50 MB of combined upload/download bandwidth each day. Every byte consumed by software updates or other overhead is taken from bandwidth that could otherwise be used to return valuable collected data to Earth.

Heterogeneous compute. The rapid developmental pace of SmallSats [20] causes new generations of SmallSats to be launched while previous nodes are still operating, meaning that constellations tend to be very heterogeneous in nature. Even the individual computers within a single SmallSat can also use different ISAs [18], further complicating software deployment and satellite management. Ensuring that programs will behave consistently is of utmost priority, as issues adapting to different ISAs have caused missions to crash [21] or explode on launch [22].

Due to their small size, SmallSats need to work with a very limited thermal and power envelopes, which restrict them low-power mobile hardware instead of high-performance datacenter counterparts [23, 24]. With limited compute available, expensive computational overhead could lead to real-time limitations being broken, or the spacecraft may run out of power during the computation. Thus, any type of software deployment on SmallSats needs to minimize both execution time and computational overhead.

Hosted multi-tenant payloads. An emerging paradigm in spacecraft design is hosted payload modules, where a launch provider hosts sensors from multiple organizations on the same spacecraft [7]. Such a model parallels multi-tenant cloud platforms, where a single bare-metal computer may host virtual machines from multiple clients. Organizations benefit from sharing the costs of assembly, verification, and launch,

which significantly decreases the cost of a spacecraft compared to launching custom-designed individual spacecraft.

This model has been proven in many NASA missions [25–28], where different institutions manage and operate the different instruments onboard a spacecraft. Commercially, many startups are also adopting this concept, including hosted payload providers such as Rocket Lab and Redwire Space [7]. Isolation between the client programs will be essential to ensure that clients' code does not interfere with one another.

However, the current state-of-the-art relies on one monolithic binary for all tenants [16]. Updates to fix bugs or add new features for a tenant are thus an extremely painful process. The current method involves the tenant sending a specification of changes to be made to the vendor, who then implements them independently [29]. While software testing is done on a replica of satellite before being uplinked, these are by no means formal specifications, and the testing is nowhere near comprehensive. In fact, one such misinterpretation of the specifications resulted in the total loss of a $125M mission [21].

Recent research has also proposed bringing the cloud computing paradigm into space to minimize the bandwidth bottleneck on ground stations as the number of active satellites grows [11, 30, 31]. This approach involves deploying datacenters in space, which will preprocess data collected by other spacecraft, before forwarding requested data back to ground stations [8]. However, the expense of launching spacecraft, combined with the limited thermal and power envelope in space [20], will require these space-bound datacenters to rely on low-power chips without many features available in the datacenter, such as hardware virtualization or IOMMU. These developments call for an efficient way to ensure safe multi-tenancy on low-power flight computers.

3 Design Requirements

Support for heterogeneous platforms. Satellite software must address the vast array of compute elements available to operators. Computers used within a constellation or exploration mission vary in clock speed, memory, storage, available co-processors, and even ISAs. With SmallSats' very limited uplink bandwidth, writing customized binaries for every configuration within a constellation will quickly become infeasible. However, traditional spacecraft software is currently written for just a single specific hardware platform [32], with ad-hoc methods for interoperability between different onboard computers [33] or SmallSat nodes [34].

Extensibility. SmallSat missions often work under constantly changing conditions and mission requirements [35]. Keeping up with such conditions often requires complex logic that cannot be expressed with legacy sequencing approaches. For example, one application involves coordinating SmallSats to take successfully higher-resolution photos of a specific location based on data from the preceding SmallSats in the chain [36]. Historically, these maneuvers were done manually, but doing so for hundreds or thousands of small satellites is infeasible. Thus, satellite software must also be extensible and seamlessly adapt to new applications.

Since ground station bandwidth is extremely limited, the research community has also proposed solutions where computations usually done on Earth would be done on the satellite instead [8, 11, 13]. As it is often hard to foresee every possible application of a satellite at its launch [33], software updates for hardware in orbit is a routine occurrence [18]. However, as flight software is a continuously-running, hard real-time process [37], updates require scheduling a service blackout while new software capabilities are added [18]. To reduce downtime from these updates, our software runtime should ensure that component changes are seamless.

Lightweight correctness and isolation. Flight software is a critical real-time system that spacecraft rely on for core functions, such as navigation and positioning or propulsion and thermal upkeep [29]. Resource starvation could also cause critical deadlines, such as a change in orbit to avoid space debris, to be missed [38]. Thus, in the compute-constrained space environment, we need to ensure that program components do not take up an inordinate amount of compute time.

Failures in flight software have often resulted in explosive mission failures [21, 22, 39]. However, as SmallSats are "wimpier" nodes than high-orbit satellites, they often do not have strong per-node fault tolerance, rather relying on failing over to the rest of the constellation to maintain uptime [40]. This still represents a significant cost to constellation operators, as each failed node still costs tens of thousands of dollars to build and launch. Thus, flight software must ensure that errors should be isolated to its component in a lightweight way.

In a multi-tenant flight software system, each tenant should not be able to access another tenant's memory space, or have errors propagate past the protection boundary. To this end, memory safety and isolation of components are a foremost priority for satellite software. Traditionally, spacecraft operators have only addressed memory safety by disallowing dynamic memory allocation in their software development guidelines. However, this approach greatly limits the extensibility of the software. Adapting modern systems approaches, such as enforcing memory and type safety [41], may provide a lightweight way to ensure isolation and error protection while also allowing the software to adapt to changing requirements.

4 Bytecode-Based Runtime

We thus propose a design that replaces the old sequencing-based hardware-dependent design with an extension-based design using a bytecode runtime. Such a design can ensure multi-tenant isolation, while keeping the safety constraints of flight software and improving deployment times.

High-level tasks run in a userspace sandbox. As shown in Figure 1, our design divides a satellite's software capabilities into high-level bytecode extensions and low-level native

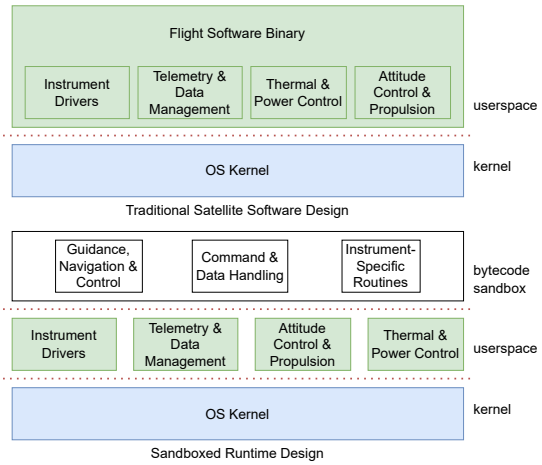

Figure 1: Extension-style satellite software design compared to traditional approach. Blue and green boxes represent native kernel and userspace code respectively, white boxes are bytecode that runs in a userspace sandbox, and the red dotted line represents a protection boundary.

code. Low-level capabilities interface directly with hardware and provide APIs for high-level capabilities to use the hardware, and thus need to be expressed in native code. On the other hand, high-level capabilities such as planning and decision-making, which do not involve hardware at all, will be isolated within a *sandbox running in userspace*.

As the vast majority of software changes in real-world spacecraft missions also modify high-level code, expressing it in a bytecode runtime can further lower update bandwidth requirements. We examined the flight software of two real-world NASA missions, containing 8.4 and 4.6 million lines of code. We found that in both repositories, ~15% of the code represents high-level behaviors, which do not directly interface with devices. Of all lines changed by software updates deployed to these spacecraft, ~85% of them were changes to the aforementioned high-level code. Therefore, we can conclude that the high-level code changes much more frequently than the "base layer" code, which remains relatively stable.

When using bytecode, the binary also no longer needs to restart after updating, minimizing service disruption [18].

Safety constraints. The naive approach of dynamically loading shared libraries introduces additional validation and safety headaches. As shared libraries essentially contain "black-box" code, allowing them to be linked into the memory space of a critical flight software process in a multi-tenant environment presents a security and correctness threat. In contrast, an isolated, verified bytecode satisfies real-time and safety constraints in flight software. Verification can ensure that the bytecode will terminate and does not access memory out-of-bounds [42]. The effort needed for the program to pass validation can be offloaded to the tenants, rather than the hosted payload provider. Bytecode also allows the provider to easily insert checkpoints to ensure that deadlines are not missed.

Easing deployments to heterogeneous constellations. Instead of uploading a different binary for each flight computer's hardware configuration, we can instead send a single bytecode that works across all configurations in the constellation, saving precious uplink bandwidth, as well as greatly simplifying the task of satellites propagating the updates peer-to-peer. As satellites travel in a train formation, every node in the constellation will always be able to communicate with at least the nodes ahead and behind it, like a linked list. This allows for far more flexibility in scheduling data transfers compared to the limited amount of uplink windows available for ground-to-space transfers.

To motivate this design, we simulate update propagation across a constellation of 150 SmallSats at an altitude of 400km equipped with state-of-the-art VHF radios [7] capable of inter-satellite communication and a ground station at 42°N latitude. Figure 3 shows the time to send a 50MB update for one to four different hardware configurations requiring unique binaries across ten trials, starting at a random time between two transmission windows. The update time increases as more configurations need to be supported, since the ground station needs to uplink each version of the update to at least one of the satellites, but it can attain line-of-sight to the constellation only a few times each day. Thus, it is more efficient to uplink the binary for one configuration at a time, since once any binary upload to a satellite completes, it can be distributed and applied to other satellites with similar configurations between uplink windows. The results show that even updating 2 configurations becomes prohibitively expensive, taking almost two days in our simulation due to the limited bandwidth. This might be a significant issue for LEO satellite operators, which may need to push changes to their satellites multiple times a day.

5 Comparison of Bytecode Runtimes

We aim to build upon an existing well-defined runtime that can benefit from community support, as building a new bytecode runtime requires vast developmental efforts. We thus evaluated how three widely-used bytecode runtimes fit our design requirements (§3). We found that across a variety of dimensions listed in Table 1, uBPF, a JIT-capable runtime for eBPF, is the most promising runtime to use as our foundation. However, significant gaps remain in uBPF's current capabilities before it is ready to be a runtime for vast arrays of wimpy satellites. We now dive into the details of our comparison.

Benchmarks. We considered three popular bytecode runtimes for embedded devices: Java on OpenJDK, Webassembly on Wasmtime, and eBPF on the uBPF userspace runtime. As

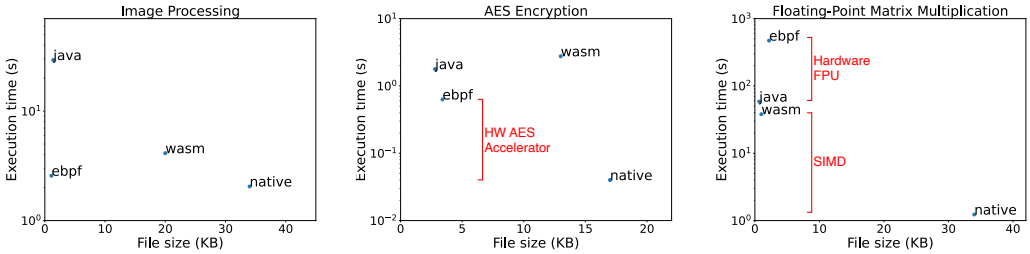

Figure 2: Comparison of execution time and binary size of common bytecode runtimes. The closer to the lower left a runtime is, the more optimal it is. Red labels show gaps in eBPF's ISA that are open challenges described in §6.

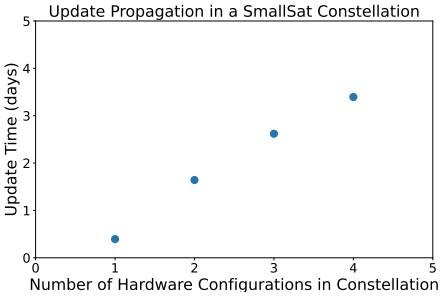

Figure 3: Time needed for a 50MB software update to uplink and propagate to a constellation of 150 SmallSats, each with 3Mbps radios and one ground station.

Feature	Native	OpenJDK	wasmtime	uBPF
Cross-ISA		✓	✓	✓
Safety & Isolation		✓	✓	✓
Termination Ensured				✓
Small Binary Size		✓		✓
Efficient on Low-Power	✓			✓
Floating-Point	✓	✓	✓	OC
SIMD Support	✓	✓		OC

Table 1: Comparison of candidate bytecodes and features important for use on satellites. Entries labeled OC are open challenges outlined in §6.

shown in Figure 2, we run three benchmarks that reflect common software onboard spacecraft. The first is a navigational workload tested on real-world spacecraft [14] which relies on integer operations across large matrices. The second runs 10M rounds of AES encryption, which ensures the integrity and security of spacecraft communication. Our final test tests 10M multiplications of a 10x10 floating-point matrix, a task underlying critical operations such as state estimation with Kalman filters, and a key component of neural networks.

Binary size (upload time). The first dimension we compared the runtimes is on the size of the binary or bytecode that would need to be uploaded to space. We found that native binaries are larger than bytecode since helper functions provided in bytecode runtimes need to be included in the native binary. Java bytecode sizes tend to be smaller than Wasm, as Java provides a large standard library that is not included in the binary. Even though a portion of the C standard library was included in both eBPF and Wasm bytecode, we find that eBPF tends to be much more compact than Wasm, since Wasm's stack machine format adds significant size to the binary.

Execution time. Native binaries unsurprisingly are much faster than JIT-compiled bytecode in terms of execution time. Wasmtime consistently had the longest executions, which previous work attributes to its stack structure [43], which adds considerable overhead to the JIT and emits suboptimal native instructions. OpenJDK's poor performance is because the JIT is triggered as a function of how frequently a code section is executed, meaning that JIT-ting happens in parallel with execution [44]. The low-power ARM CPU used in this benchmark was unable to effectively handle this compute load. In contrast, uBPF efficiently precompiles bytecode before execution, saving expensive interpretation cycles for integer operations [45], as in the image processing task. However, its performance suffers when programs need to use more specialized pipelines, which we discuss below.

Hardware support. Due to their open-source nature, all three bytecodes have well-supported interpreters for common spacecraft ISAs, such as ARM, x86, and PowerPC. However, we note that Java may not be suitable for low-power devices due to the high JIT overhead. Similarly, Wasm is expensive to translate to native ISAs, as it uses a stack machine architecture that differs from the register-based native ISAs. This adds inescapable overhead, as the JIT must run an expensive register allocation pass when translating to native code [46], and the register allocation may be less efficient than one generated by a full compiler such as LLVM. In comparison, eBPF's ISA is register-based and easily mappable to all host ISAs, which makes JIT translation fast and efficient.

The importance of architecture-aware programming is highlighted by the results of the encryption and matrix multiplication benchmarks. The native code was able to take advantage of ARM's NEON pipelines and encryption accelerators, which significantly reduces its execution time. While both Java and Wasm could take advantage of the host FPU in the matrix multiplication benchmark, eBPF's ISA has no floating point extension. This required us to write a software floating point library, which executes floating point operations as

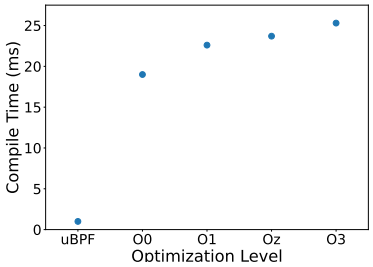

Figure 4: Time for uBPF and LLVM to compile eBPF bytecode for an image processing algorithm to native RISC-V, at various optimization levels.

multiple ALU instructions. This requires many more cycles per operation, and incurs a heavy execution time penalty.

Safety and isolation. Unlike native binaries, all three runtimes provide a meaningful measure of isolation with built-in protections. As for memory safety, both Wasm and Java assume that code is untrusted. They rely on expensive runtime checks to safely run untrusted code [47], but many potential exploits still remain [48]. While there has been some previous work in runtime safety solutions for bytecodes [9, 49], they usually rely on interpreting bytecode to ensure memory safety, which incurs a significant performance impact.

Rather than relying on runtime protections, eBPF employs a verifier which ensures that all pointer arithmetic is valid and in bounds before program execution. The verifier also makes sure that eBPF programs will terminate, which rules out a destructive class of bugs for spacecraft. Verification can occur on the ground before the functions are uploaded to the satellites, minimizing the onboard performance impact [50].

Results summary. By using a RISC ISA that closely mirrors popular hardware ISAs, eBPF achieves small binary sizes while remaining fast on low-power devices, outclassing Wasm and Java. eBPF's verifier also guarantees safety and termination properties that other runtimes do not. Though it currently lacks support for floating-point and SIMD pipelines, support for these can be added to eBPF by extending the ISA. eBPF thus has the potential to become an ideal bytecode for satellite software systems, though challenges remain that we outline in §6.

6 Open Challenges

While an eBPF userspace runtime is a promising foundation for a satellite runtime, significant open challenges remain.

Optimizing instruction selection. To be useful for spacecraft control, eBPF must support native floating-point calculations. Though compilers can automatically select an optimal instruction set for native binaries using semantics in the source code such as live variables and loop invariants, these semantics are lost when the code is translated to a low-level bytecode or native assembly. Thus, JIT compilers do not have enough information to translate bytecode to more optimized pipelines without reverse-engineering these semantics with expensive algorithms. This can be seen in Figure 4, where directly translating instructions is far more efficient than using traditional compiler techniques, even without optimization.

An open research question involves how to incorporate metadata into bytecode to provide the JIT compiler with additional insight into the original semantics. To meet power constraints, we need to minimize the amount of operations the JIT compiler does during the translation stage. However, we are also limited by the amount of metadata we can include, due to the very limited uplink bandwidth available. This design space thus involves a trade-off between the level of detail we can include in the bytecode while still remaining storage-efficient. For vectorization, a potential method could cache addresses of vectorization candidates and add inner loops to mimic SIMD form, minimizing the work the JIT needs to do.

Scheduling multi-tenant jobs with real-time constraints. eBPF needs to be adapted to the real-time environment spacecraft operate in. Space is full of hard real-time deadlines, such as time-limited transmission windows and short flyover times. With multiple tenants with varying deadlines to deal with and a low-power computer hosting it all, some lightweight, performant fair-share or QoS scheduler will need to be implemented, that can efficiently and fairly allocate scarce resources such as CPU, memory, network bandwidth and energy.

This scheduling may also need to be distributed across multiple satellites in a constellation. For example, the same transmission window or flyover target can be visible to successive satellites in a train formation at different times. Most workloads will simply require at least one, but not all of the satellites to do work during the flyover [36]. Furthermore, each node can only communicate with a limited number of their peers, and higher failure rates in space may mean that these peers may not stay consistent. Thus, developing efficient inter-satellite scheduling for the intermittently connected space environment will be essential to enabling the next generation of satellite compute.

Radiation hardening. Elevated radiation levels in space can cause silent data corruption (SDC) on spacecraft computers [51]. To remain performant but cheap, SmallSat compute elements often lack the radiation hardening afforded to their high-orbit counterparts [7]. Thus, a runtime should also include software protections against radiation-induced SDC, a relatively unexplored area in systems research [52].

eBPF's safety properties partially addresses this issue. For example, the termination guarantee can detect SDCs that cause hangs, while the runtime can detect corruptions in pointer addresses as invalid accesses. However, SDC in execution data could still result in incorrect results, and triplicating the runtime, though effective, but may be too expensive computationally. Detecting and fixing SDCs in a single-threaded manner, all without breaking real-time guarantees, remains an open challenge.

References

[1] I. E. Mladenova et al. "Evaluating the operational application of SMAP for global agricultural drought monitoring". In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 12.9 (2019), pp. 3387–3397.

[2] C. Boshuizen et al. "Results from the Planet Labs Flock Constellation". In: (2014).

[3] L. Scheck, M. Weissmann, and L. Bach. "Assimilating visible satellite images for convective-scale numerical weather prediction: A case-study". In: *Quarterly Journal of the Royal Meteorological Society* 146.732 (2020), pp. 3165–3186.

[4] F. Michel et al. "A first look at starlink performance". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 130–136.

[5] *Hughes and OneWeb Announce Agreements for Low Earth Orbit Satellite Service in U.S. and India*. https://www.hughes.com/resources/press-releases/hughes-and-oneweb-announce-agreements-low-earth-orbit-satellite-service-us. 2021.

[6] H. Jones. "The recent large reduction in space launch cost". In: 48th International Conference on Environmental Systems. 2018.

[7] B. Yost and S. Weston. *State-of-the-art small spacecraft technology*. Tech. rep. 2024.

[8] N. Bleier et al. "Space Microdatacenters". In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 2023, pp. 900–915.

[9] R. Liu, L. Garcia, and M. Srivastava. "Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices". In: *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2021, pp. 94–105.

[10] B. Denby et al. "Kodan: Addressing the computational bottleneck in space". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2023, pp. 392–403.

[11] D. Bhattacherjee et al. "In-orbit computing: An outlandish thought experiment?" In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 2020, pp. 197–204.

[12] S. Kuhn. "Thermal is the Plan the Plan is Death: Deployment of the Mars 2020 On-Board Planner". In: *2024 IEEE Aerospace Conference*. IEEE. 2024, pp. 1–21.

[13] B. Tao et al. "Known Knowns and Unknowns: Near-realtime Earth Observation Via Query Bifurcation in Serval". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 809–824.

[14] J. Nash et al. "Censible: A Robust and Practical Global Localization Framework for Planetary Surface Missions". In: *IEEE International Conference on Robotics and Automation* (2024).

[15] V. Z. Sun et al. "Evolution of the Mars 2020 Perseverance Rover's Strategic Planning Process". In: *2024 IEEE Aerospace Conference*. IEEE. 2024, pp. 1–16.

[16] R. Bocchino et al. "F Prime: an open-source framework for small-scale flight software systems". In: (2018).

[17] T. H. Dixon. "An introduction to the Global Positioning System and some geological applications". In: *Reviews of geophysics* 29.2 (1991), pp. 249–276.

[18] A. Badshah, N. Morris, and M. Monson. "Over-The-Vacuum Update – Starlink's Approach for Reliably Upgrading Software on Thousands of Satellites". In: *Small Satellite Conference*. Aug. 2023.

[19] P. A. Ilott. "Communications with Mars: A Brief and Informal HIstory". In: *2021 Space-Terrestrial Internetworking workshop (STINT)* (2021).

[20] W. A. Powell. "High-performance spaceflight computing (hpsc) project overview". In: *Radiation Hardened Electronics Technology Conference (RHET) 2018*. GSFC-E-DAA-TN62651. 2018.

[21] M. I. Board. *Mars Climate Orbiter Mishap Investigation Board Phase I Report November 10, 1999*. 1999.

[22] M. Dowson. "The Ariane 5 software failure". In: *ACM SIGSOFT Software Engineering Notes* 22.2 (1997), p. 84.

[23] J. Murphy et al. "Deploying Machine Learning Anomaly Detection Models to Flight Ready AI Boards". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 6828–6836.

[24] E. R. Dunkel et al. "Benchmarking deep learning models on myriad and snapdragon processors for space applications". In: *Journal of Aerospace Information Systems* 20.10 (2023), pp. 660–674.

[25] G. Chin et al. "Lunar reconnaissance orbiter overview: The instrument suite and mission". In: *Space Science Reviews* 129 (2007), pp. 391–419.

[26] J. P. Grotzinger et al. "Mars science laboratory mission and science investigation". In: *Space science reviews* 170 (2012), pp. 5–56.

[27] K. A. Farley et al. "Mars 2020 mission overview". In: *Space Science Reviews* 216 (2020), pp. 1–41.

[28] R. T. Pappalardo et al. "Science overview of the Europa clipper mission". In: *Space Science Reviews* 220.4 (2024), p. 40.

[29] D. Dvorak. "NASA study on flight software complexity". In: *AIAA infotech@ aerospace conference and AIAA unmanned... unlimited conference*. 2009, p. 1882.

[30] Y. Michalevsky and Y. Winetraub. "WaC: SpaceTEE-Secure and Tamper-Proof Computing in Space using CubeSats". In: *Proceedings of the 2017 Workshop on Attacks and Solutions in Hardware Security*. 2017, pp. 27–32.

[31] S. K. Johnson et al. "Gateway–a communications platform for lunar exploration". In: *38th International Communications Satellite Systems Conference (ICSSC 2021)*. Vol. 2021. IET. 2021, pp. 9–16.

[32] G. E. Reeves and J. F. Snyder. "An overview of the Mars exploration rovers' flight software". In: *2005 IEEE International Conference on Systems, Man and Cybernetics*. Vol. 1. IEEE. 2005, pp. 1–7.

[33] V. Verma et al. "Enabling Long & Precise Drives for The Perseverance Mars Rover via Onboard Global Localization". In: *2024 IEEE Aerospace Conference*. IEEE. 2024, pp. 1–18.

[34] J. A. Gutierrez Ahumada, K. Doerksen, and S. Zeller. "Automated fleet commissioning workflows at Planet". In: (2021).

[35] J. Mason et al. "Orbital debris–debris collision avoidance". In: *Advances in Space Research* 48.10 (2011), pp. 1643–1655.

[36] Z. Cheng et al. "EagleEye: Nanosatellite constellation design for high-coverage, high-resolution sensing". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 117–132. ISBN: 9798400703720. DOI: 10.1145/3617232.3624851. URL: https://doi.org/10.1145/3617232.3624851.

[37] T. Durkin. "What the Media Couldn't Tell You About Mars Pathfinder". In: *Robot Science & Technology* 1 (1998).

[38] N. G. Leveson. "Role of software in spacecraft accidents". In: *Journal of spacecraft and Rockets* 41.4 (2004), pp. 564–575.

[39] A. Albee et al. "Report on the loss of the Mars Polar Lander and Deep Space 2 missions". In: (2000).

[40] J. Cappaert et al. "Constellation modelling, performance prediction and operations management for the spire constellation". In: (2021).

[41] A. Levy et al. "Multiprogramming a 64kb computer safely and efficiently". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 234–251.

[42] E. Gershuni et al. "Simple and precise static analysis of untrusted Linux kernel extensions". In: PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1069–1084. ISBN: 9781450367127. DOI: 10.1145/3314221.3314590. URL: https://doi.org/10.1145/3314221.3314590.

[43] Y. Yan et al. "Understanding the performance of webassembly applications". In: *Proceedings of the 21st ACM Internet Measurement Conference*. IMC '21. Virtual Event: Association for Computing Machinery, 2021, pp. 533–549. ISBN: 9781450391290. DOI: 10.1145/3487552.3487827. URL: https://doi.org/10.1145/3487552.3487827.

[44] T. Suganuma et al. "Overview of the IBM Java just-in-time compiler". In: *IBM systems Journal* 39.1 (2000), pp. 175–193.

[45] S. Kubica and M. Kogias. "μBPF: Using eBPF for Microcontroller Compartmentalization". In: *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*. 2024, pp. 23–29.

[46] K. Zandberg et al. "Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers". In: *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. Middleware '22. Quebec, QC, Canada: Association for Computing Machinery, 2022, pp. 161–173. ISBN: 9781450393409. DOI: 10.1145/3528535.3565242. URL: https://doi.org/10.1145/3528535.3565242.

[47] J. Dejaeghere et al. "Comparing Security in eBPF and WebAssembly". In: *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*. eBPF '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 35–41. ISBN: 9798400702938. DOI: 10.1145/3609021.3609306. URL: https://doi.org/10.1145/3609021.3609306.

[48] C. Disselkoen et al. "Position paper: Progressive memory safety for webassembly". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2019, pp. 1–8.

[49] M. Research. *DeviceScript - TypeScript for Tiny IoT Devices*. https://github.com/microsoft/devicescript. 2022.

[50] M. Craun, A. Oswald, and D. Williams. "Enabling eBPF on Embedded Systems Through Decoupled Verification". In: *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*. eBPF '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 63–69. ISBN: 9798400702938. DOI: 10.1145/3609021.3609299. URL: https://doi.org/10.1145/3609021.3609299.

[51] E. Normand. "Single-event effects in avionics". In: *IEEE Transactions on nuclear science* 43.2 (1996), pp. 461–474.

[52] H. Wang et al. "Mars Attacks! Software Protection Against Space Radiation". In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 2023, pp. 245–253.