Vivid: An Operating System Kernel for Radiation-Tolerant Flight Control Software

by

Cel Andromeda Skeggs

Bachelor of Science in Computer Science and Engineering
Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

May 2022

Signature of Author: _____
Department of Electrical Engineering and Computer Science
May 15, 2022

Certified by: _____
M. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

Accepted by: _____
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Vivid: An Operating System Kernel for Radiation-Tolerant Flight Control Software

by

Cel Andromeda Skeggs

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2022 in Partial Fulfillment of the
Requirements for the Degree of Master of Engineering in
Electrical Engineering and Computer Science

ABSTRACT

This thesis considers the challenge of defending flight software from radiation errors without a radiation-hardened processor. A new real-time operating system, *Vivid*, explores the use of redundant multithreading to protect critical software components from radiation errors, and offers new abstractions to reduce the number of single points of vulnerability in the system. It introduces a static component initialization system for C, which eliminates most runtime initialization steps from the operating system and flight software. It introduces a partition scheduler based on *execution clips*, which ensures that software components always start from a safe state, and it protects the system's safe state using a pair of memory scrubbers. Vivid introduces *voting ducts*, an inter-process communication primitive for redundant multithreading that eliminates single points of vulnerability from the voting process. Finally, it defines a *sequence of repair* that ultimately grounds the correct operation of all components in the system's software in a hardware watchdog.

To demonstrate the applicability and effectiveness of Vivid, this thesis introduces *Swivel*, a testbench spacecraft, and describes *SwivelFSW*, which is the implementation of flight software that meets Swivel's behavioral requirements, and *SwivelSim*, which is the simulation of Swivel's avionics. Next, this thesis introduces *Hailburst*, a system for efficient processor emulation and radiation fault injection, and uses it to evaluate Vivid's radiation tolerance through a series of accelerated radiation injection trials. In the tested configuration, Vivid tolerates approximately 149 out of every 150 injected radiation faults without any observed requirement failures, and recovers from the remaining 1 out of 150 radiation faults within at most 2.05 seconds of recovery time in the worst observed case. Because some of Vivid's defenses appear to be more effective than others, and some may be counterproductive, this thesis discusses future work that would be required before Vivid's abstractions could be applied to real-world flight software.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor

Acknowledgements

The past sixteen months have been challenging, even beyond the regular difficulty of a Master's program. The COVID-19 pandemic, already ongoing, continues to stretch interminably into the future, and affects every decision I make. Geopolitical tensions have risen, both here and around the world: Russia is invading Ukraine, unchecked industrial pollution continues to upset our world's climate and plunge us towards a dangerous future, and lawmakers and public figures across the United States continue to threaten the legal and physical safety of people like me.

Because of these challenges, I have a great deal of gratitude to share for too many people to list here for helping me survive and sometimes even thrive in this difficult period of history. I am extremely thankful for the endless support I have received from Frans, my advisor, whose active mentorship made this thesis possible. I also owe a great deal to Beth, my mother; when I needed to go into major surgery on short notice midway through this semester, she set aside her obligations to fly across the country and help me stumble through the hospital and the first weeks of my recovery; I also owe much gratitude to the rest of my friends that assisted with my recovery even when it was inconvenient to them.

I very much appreciate the support I have received throughout the whole of my degree program from the rest of my immediate and extended family, and I value greatly the companionship, advice, and common sense that my dear network of friends, near and far, have provided through this whole period of my life. I also harbor a great deal of appreciation for everyone else in my lab, PDOS, whose camaraderie, ideas, and knowledge helped me grow more confident in and cognizant of myself and my research over the course of the past year and a half, and I am very thankful to all of the mentors and coworkers at JPL that I worked with before entering this program, who made my interest in this area of research possible.

Table of Contents

1. Introduction

This Master of Engineering thesis seeks to help flight computers on robotic spacecraft better tolerate the intense levels of radiation that they experience in space without requiring that they employ specialized radiation-hardened processors and memory. This thesis introduces five new systems: *Vivid*, a hard-real-time operating system designed for building radiation tolerant applications on unsafe hardware; *Hailburst*, a fault injection and processor emulation framework for measuring the effects of injecting accelerated radiation faults into spacecraft software; *Swivel*, a testbench spacecraft with requirements representative of realistic requirements for flight systems; *SwivelFSW*, an implementation of flight software for Swivel that demonstrates the usability and effectiveness of Vivid; and *SwivelSim*, an avionics simulation for Swivel that continuously tests the spacecraft against its requirements.

1.1. Motivation

Earth's magnetosphere protects terrestrial electronic systems from the high levels of radiation that are present in space, which can temporarily or permanently damage electronic systems. Robotic spacecraft, including satellites and deep-space probes, are especially vulnerable to radiation damage, as they rely on onboard computers for their basic ability to function. If a spacecraft's flight computer malfunctions, the spacecraft may be lost forever: it is generally impossible (or at least infeasible) to send anyone to repair it, and if the spacecraft cannot communicate with mission control, engineers on Earth will not have the data to diagnose the problem, nor the ability to conduct any repairs (such as resets) that may be commanded remotely.

1.2. State of the Art

Because every single robotic spacecraft launched from Earth is a significant investment, space agencies like NASA spare little expense to defend against the possibility that a spacecraft might fail prematurely and be lost forever. Normally, engineers defend flight computers against the risk of radiation damage or disruption by using radiation-hardened processors, like the BAE RAD750, whose circuitry is physically redesigned to resist high levels of radiation. [1] [2] Unfortunately, these specialty devices lag behind their unhardened consumer equivalents in performance, mass, cost, and power consumption. [1] [3] [4] If new research can establish a way to effectively defend off-the-shelf flight computers from radiation, it may allow engineers to design more capable software for robotic spacecraft with lower mass and power requirements.

One common approach for defending electronic systems from unexpected faults is to use *triple-modular redundancy* (TMR). [5] In a TMR design, a system includes three replicas of the critical components that all receive the same inputs and vote on every set of outputs. In the absence of radiation, all three replicas remain perfectly synchronized, and agree on exactly what output they should produce. When radiation causes a single fault, it will damage only one of the replicas, which may make that replica misbehave and start producing the wrong output. However, since the other two replicas will still agree with each other, the majority vote of the three preliminary outputs will still result in the correct final output. As long as radiation damages only one replica, and the system can repair each damaged replica before the next radiation fault occurs, the system will be able to continue operating seamlessly through an unlimited number of radiation faults. Because radiation faults occur infrequently, on the order of a few times per day (though this varies significantly based on radiation environment and specific equipment), there is generally plenty of time for the system to repair each damaged replica. [6]

Ordinarily, engineers implement triple-modular redundancy in hardware; when they implement it in software by switching rapidly between the software for different replicas, the term is *redundant multithreading*. Existing work on redundant multithreading focuses on protecting only software applications and assumes that some other approach will protect the underlying operating system. [7]

Choice of operating system is already even more essential for spacecraft computers than terrestrial computers. Flight software generally has strict subsecond deadlines to perform critical spacecraft functions, like executing the guidance algorithms that control actuators like thrusters and reaction wheels; if the software does not perform those critical functions on time, the spacecraft may malfunction catastrophically, such as by spinning out of control and breaking apart. Therefore, engineers ordinarily build flight software on top of hard-real-time operating systems, like VxWorks, INTEGRITY, RTEMS, or FreeRTOS. [1] [8] [9] These hard-real-time operating systems provide operating system abstractions similar to the abstractions provided by general-purpose operating systems like Linux or Windows, such as tasks, semaphores, queues, file systems, and dynamic memory, except that the abstractions are carefully implemented to eliminate unexpected or uncertain delays that could cause the software to exceed its strict deadlines.

Because hard real-time operating systems ordinarily run on radiation-hardened flight computers, their designs do not account for radiation concerns. This means that redundant multithreading applications running on hard-real-time operating systems will be vulnerable to radiation errors in the underlying operating system.

1.3. New Approach

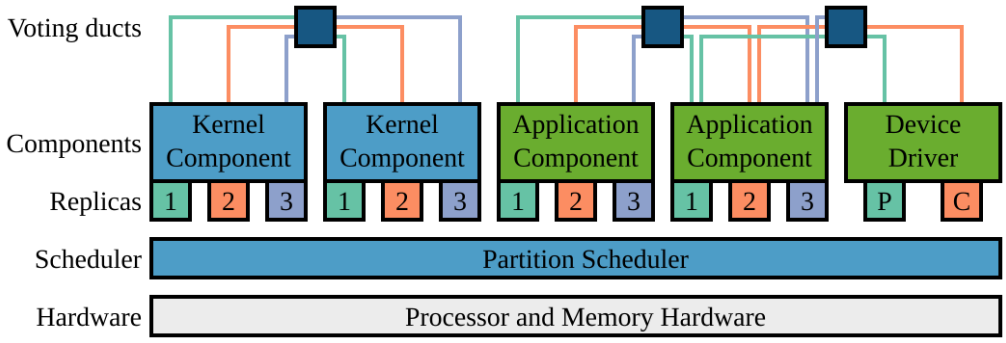


Figure 1-1: Vivid System Structure

This thesis introduces a new hard-real-time operating system, *Vivid*[1], designed to run on single-core consumer ARM processors without any hardware defenses against radiation. Vivid, shown in Figure 1-1, departs from traditional real-time operating system designs in two main ways: First, Vivid uses the microkernel approach of separating out as many features from the kernel as possible into small independent modules, leaving the partition scheduler as the single fully privileged kernel component. Second, it implements a new set of operating system abstractions targeted for defending applications with redundant multithreading. These include *voting ducts*, which allow replicated applications to communicate with each other without any runtime kernel support or single points of vulnerability to radiation. (Most microkernels, by contrast, need to provide some runtime kernel support for inter-process communication.) In many cases, such as for parts of its watchdog driver, Vivid's kernel components are also replicated, just like its application components.

Vivid seeks to seamlessly mitigate almost all of the possible radiation faults that flight software might experience. It cannot seamlessly mitigate *all* possible radiation faults, because

[1] The name "Vivid" refers to the definition of the word meaning "full of life or vitality." This references the operating system's ability to thrive in environments uninhabitable by other operating systems.

failures in critical parts of the operating system (like the scheduler) can disrupt the system's ability to function beyond the ability of software to repair the problem. In these cases, where visible failures are inescapable, Vivid uses a watchdog to ensure that the flight software will recover within a short, bounded interval of time and resume executing, possibly after a partial or complete reset of the system's state.

To evaluate whether Vivid can actually satisfy these reliability goals, this thesis presents five new systems. The first system is a prototype implementation of Vivid that runs on an emulated ARM platform. The second system is *Swivel*[2], a hypothetical design for a simple testbench spacecraft with a list of representative behavior requirements. The third system is *SwivelFSW*[3], which is an implementation of flight software on Vivid for the Swivel spacecraft, and the fourth system is *SwivelSim*[4], which is a simulation of Swivel's avionics equipment that can evaluate the flight software against the list of behavior requirements from the design. The fifth system is *Hailburst*[5], an efficient and deterministic tool for injecting radiation faults into an emulated processor. This thesis uses Hailburst and SwivelSim to evaluate the ability of SwivelFSW to satisfy its requirements in the presence of simulated radiation, and to determine how effective Vivid's design actually is at defending flight software from radiation errors.

1.4. Contributions and Results

This thesis makes several contributions:

1. The design of Vivid, a hard-real-time operating system for building radiation-tolerant flight software using redundant multithreading on single-core processors. It includes:

[2] The name "Swivel" refers to the least essential element of a staffed avionics testbed: the swivel chair.
[3] SwivelFSW is short for "Swivel Flight Software."
[4] SwivelSim is short for "Swivel Simulation."
[5] The name "Hailburst" refers to the accelerated fault injection features of the simulation system, and how they involve dumping a large number of potentially damaging particles over a wide area in a short period of time.

a. A compile-time component initialization system for C, which allows engineers to build complex low-level software applications without dynamic resource allocation.
b. A memory scrubber that leverages the component initialization system to protect flight software's minimum safe state.
c. The *execution clip* abstraction for Vivid's partition scheduler, which allows seamless correction of radiation vulnerabilities in implicit register and stack state.
d. The *voting duct* concurrency primitive that allows redundant multithreading deployments to avoid single points of vulnerability in their voting stage.
e. The *voting notepad* for voting on redundant-multithread mutable states by treating feedforward state as a loopback message.
f. The *prepare/commit* device driver design for radiation safety when devices are unreplicated.
g. A *sequence of repair* to reliably ground the recovery of a complete flight software application in a hardware watchdog.

2. The implementation of a prototype of Vivid, which includes:
    a. *Siren*, an extended C preprocessor to facilitate instantiating software components at compile time.
    b. A *replication linker*, which allows object code in an application binary to be selectively replicated to avoid single points of vulnerability in shared code.
3. Hailburst, an efficient and deterministic tool for automatically injecting radiation faults into a QEMU emulation of a spacecraft processor. Compared to existing fault injection tools, it contributes:

    a. An approach to precisely target the injection of faults into QEMU without loss of emulation performance through the use of QEMU's virtual time mechanism and an emulator patch.

4. The design of an avionics testbench spacecraft, Swivel, which contributes:
    a. A set of devices with behavior requirements that are simpler than, but representative of, the requirements of real robotic spacecraft.
    b. The *FakeWire* protocol for tunneling a SpaceWire link over a serial port interface.
5. The design and implementation of SwivelSim, a simulation for Swivel, which contributes:
    a. A tool for continuously generating test inputs, and continuously evaluating actual spacecraft behavior, to continuously report on the spacecraft's adherence to or deviation from the correct behavior indicated by Swivel's requirements, and how it changes over the course of a simulation.
    b. The *timesync protocol* that connects a processor emulated in QEMU to a spacecraft bus simulated in a separate program while maintaining complete determinism in the combined simulation.
6. The design and implementation of SwivelFSW, the flight software for Swivel. It contributes:
    a. A demonstration of implementing flight software using Vivid's abstractions.
    b. A spacecraft bus topology using virtual network switches that treats the flight software as an extension of the network architecture, so that each software component is treated as its own device on the spacecraft bus.

c. A flight software topology that allows several modules to send and receive command and telemetry messages without runtime registration.

d. A set of device managers that demonstrate how engineers can implement state machines in redundant-multithreaded software components.

e. A network device driver that demonstrates Vivid's prepare/commit driver design.

f. A schedule that enables the efficient flow of requests and responses across a spacecraft bus without compromising on strict time partitioning.

7. A preliminary evaluation of Vivid's ability to defend flight software against radiation faults, which indicates:

    a. Vivid is successful at seamlessly recovering from the majority of injected radiation faults without violating any behavior requirements. However, in the remaining minority of about one out of every 150 faults, it does violate one or more behavior requirements for a short time before recovering.

    b. When radiation faults lead to visible failures, Vivid's defenses consistently allow Swivel's flight software to recover back to working order within just over two seconds, which is the expected worst-case recovery time given the watchdog configuration.

    c. Vivid's other defense mechanisms appear to be more effective than the redundant multithreading system itself. Further evaluation and analysis will be required to determine which of Vivid's design features are most essential, and iterations on Vivid's design may improve the effectiveness of the redundant multithreading system.

1.5. Source Code Overview

The source code for all of the components in this thesis is available online on GitHub. The bulk of the code for Vivid, Hailburst, Swivel, SwivelFSW, and SwivelSim is available under https://github.com/celskeggs/hailburst, and is not yet separated neatly into separate modules. (The patches to QEMU are available at https://github.com/celskeggs/qemu.) Table 1-2 provides an overview of the number of lines of code written as part of this thesis.

Lines of Code	Source Module
6,519	SwivelFSW
8,685	SwivelSim
4,997	Vivid
1,646	Hailburst
4,538	Analysis Tools
1,059	Linux support code[6]

Table 1-2: Source Code Breakdown

1.6. Outline

This thesis proceeds as follows: Section 2. Related Work describes the existing work in defending spacecraft avionics from radiation errors and injecting faults into avionics simulations; Section 3. Design of the Vivid Kernel explains the design of Vivid and all of its abstractions and safety properties; Section 4. Implementation of the Vivid Kernel explains the prototype implementation of Vivid; Section 5. The Swivel Spacecraft describes Swivel's design and requirements; Section 6. The SwivelFSW Flight Software describes the SwivelFSW flight

[6] The Linux support code is code that can be compiled together with SwivelFSW to make a version of the flight software that runs on embedded Linux, instead of Vivid. It is not a major component described in this thesis.

software; Section 7. The SwivelSim Avionics Simulation describes the simulation of Swivel's avionics; Section 8. The Hailburst Fault Injection System describes Hailburst's design and implementation; Section 9. Evaluation explains the results of the preliminary evaluation of Vivid's reliability; and, finally, Section 10. Future Work and Summary discusses directions for future work on Vivid's design and evaluation.

2. Related Work

Section 2.1. Radiation Defense discusses existing approaches to defending flight software against radiation errors, and how Vivid differs from them. Section 2.2. Fault Injection discusses existing approaches to fault injection, and how Hailburst differs from them.

2.1. Radiation Defense

Defenses for radiation faults fall into four categories: 1) hardware implementation defenses, such as radiation-hardened computers, 2) software implementation defenses, such as special compilers that insert redundant computations, 3) hardware architecture defenses, such as combining multiple separate processors in a redundant configuration, and 4) software architecture defenses, such as replicating software into multiple running processes on a single processor. Vivid falls into the software architecture defense category.

Hardware implementation defenses, such as those used for the BAE RAD750, are well-established, but radiation-hardened computers have lower performance, higher cost, higher power consumption, and higher mass requirements compared to off-the-shelf alternatives. [1] [3] [4] [10] Vivid, like many other research projects, seeks to help mitigate these challenges by defending off-the-shelf hardware.

The ASCOT testbed on the ARGOS satellite used two software implementation defenses, EDDI (Error Detection by Duplicated Instructions) and CFCSS (Control-Flow Checking by Software Signatures), and one software architecture defense, a memory scrubber, to defend an off-the-shelf processor from radiation errors. [11] Vivid also uses a memory scrubber, but not instruction-level code modifications; while these approaches improved the reliability of the

off-the-shelf processor used in ASCOT, the experiment still experienced a significant number of silent data corruptions, hangs, and crashes, suggesting that other defenses will be required to effectively defend a system.

CEDA is an example of a newer iteration of the same control-flow checking approach from the ASCOT testbed, of which it is only one of many. [12] It aims to maximize the number of errors that can be detected while minimizing the performance overhead of the technique. It only detects radiation errors, and does not correct them. Vivid, by contrast, accepts a high performance penalty by using Redundant Multithreading in exchange for detecting and correcting as many faults as possible.

The CALIPSO mission used a quadruple-replicated off-the-shelf processor as its primary flight controller. [6] This is an example of a hardware architecture defense; it did not require significant modifications to the flight software, but instead relied on hardware voting to protect against processor errors, and hardware-based EDAC (error detection and correction) to protect against memory errors. Vivid, by contrast, attempts to operate correctly even when the processor and main memory are unprotected from radiation faults.

The SpaceCube RHBSW (Radiation Hardening By Software) techniques include checkpoint-and-restart, control-flow assertions (like in CEDA or ASCOT), and watchdog timers. [5] Vivid also implements the use of a watchdog timer, but doesn't support checkpoint-and-restart techniques, which are appropriate for scientific computing software, not flight control software; attempting to roll back a guidance software control loop to a previous state would violate its real-time requirements.

The ADDAM (Adaptive Dependable Distributed Aerospace Middleware) platform is a system for distributing fault-tolerant software, which supports the hardware architecture defense

of distributing software across computers. [13] However, ADDAM depends on the Raft consensus algorithm to coordinate separate machines, which tolerates only disconnections and crashes, not silent data corruptions. [14] Further, it is not clear how ADDAM might support hard-real-time requirements, nor whether it intends to. By contrast, Vivid is a hard-real-time operating system, and intends to prevent silent data corruptions entirely.

The AIR operating system is a real-time operating system based on time and space partitioning, which is an example of a software architecture defense. [15] It separates different parts of flight software by function, and places them in independent partitions with independent operating system kernels. Like Vivid, it switches between partitions only on a sequence of fixed time slices; however, it is not intended to address radiation challenges, and its radiation defense is limited to protecting partitions from errors in other partitions.

Romain is an operating system service for the Fiasco.OC microkernel, which transparently replicates a binary application using Redundant Multithreading. [7] It is an example of a software architecture defense similar to Vivid; however, Romain is not designed for real-time applications, which makes it inapplicable to flight control software, and it does not defend itself or the Fiasco.OC microkernel against radiation errors, so there is a significant hole in its possible reliability.

Theseus is an operating system written in Rust that improves the modularity of an operating system by reducing the amount of state held for one component by another. [16] Among other goals, Theseus uses this structure to help it recover from kernel-level hardware faults, such as radiation errors. Like Vivid, Theseus splits the operating system into many small components that hold minimal state, but Theseus relies on Rust's language safety features for

recovery, which can be undermined by radiation errors, and it does not include any specific support for executing redundant software applications.

2.2. Fault Injection

Existing fault injection tools are difficult to use to evaluate the reliability of Vivid and its flight software, for a number of reasons. First, many fault injection tools have not been published, depend on unavailable proprietary tools, depend on the accessibility of physical devices to test, or no longer compile or function correctly on modern operating systems due to a lack of maintenance. Second, many fault injection tools feature designs that limit simulation performance compared to that of the underlying emulator, which makes them inapplicable to full-system simulation. Third, some fault injection tools bias the times or locations where they inject faults; this skews the fault injection results in unpredictable ways. Hailburst avoids these failure modes to the extent possible.

The majority of fault injection tools do not use software simulations. Some systems require a physical copy of the processor and inject faults through the use of accelerated radiation sources, pulsed lasers, or JTAG ports. Other systems use FPGAs, either to allow access to processor state, or to instrument processor circuitry with fault injection capabilities. Quinn et al provides an overview of several varieties of these tools. [17] Physical hardware-based solutions can be very efficient, because they can run at full realistic execution speeds, but they are very opaque: it is possible only to instrument physical devices to the extent allowed by their debugging interfaces. By contrast, appropriate changes to the source code of a software simulation will allow any aspect of a software simulation to be instrumented. FPGA systems are

easier to instrument than physical systems, but depend on the availability of realistic processor designs that can be instantiated in the gate array; many processor designs are proprietary and not generally available. Hailburst uses a software simulation both for versatility and to avoid the need for specialized hardware.

FEMU is an instruction-level fault simulation tool based on QEMU. [18] QEMU is a virtual machine host that uses dynamic binary translation to efficiently virtualize guest machines. [19] FEMU simulates permanent faults, intermittent faults, and transient faults by forcing QEMU into single-step mode, which allows it to modify registers and memory after every single instruction. QEMU's standard single-step mode, measured independently of FEMU's actual implementation, degrades the execution speed of QEMU significantly; the performance of simulating the Linux kernel boot process drops by nearly 65% in single-step mode, not even counting the impact of running FEMU's fault injection code on every single instruction.

DRSEUS is an instruction-level fault simulation tool based on Simics, a proprietary full-system simulation tool. [20] It takes advantage of the Simics checkpoint file format to dump system states to disk, mutate checkpointed state to inject bitflips, and resume execution from the mutated state. Because Simics is proprietary, it was not readily usable for this thesis, and because simulation checkpoint formats are generally highly specialized and not easily inspected, the same approach is difficult to apply to other simulation systems. QEMU, for example, stores its checkpoints as an extension to the QCOW2 disk format, but the details of the VM state in the snapshots are not documented, which makes it difficult to inject bitflips into the state without potentially corrupting the data structures of QEMU itself. [21]

FIG-QEMU is a fault injection platform based on QEMU that takes advantage of the existing debugging interface between QEMU and GDB to inject single-bit errors into the guest

virtual machine. [22] This is an efficient solution, except that it only supports injecting a single error per execution; in a complex piece of off-the-shelf software, like Linux, most of these errors will not produce any observable errors, so the limitation of only injecting one fault per execution makes this tool prohibitively slow, even though the actual execution is fast. In addition, FIG-QEMU uses clock-time offsets to decide when to inject faults; the interruption of an executing QEMU process from the outside biases the faults injected: moments when QEMU pauses (such as to translate additional binary code) will receive disproportionately many faults, whereas instructions in the middle of long translated blocks will receive disproportionately few.

FAIL* is a fault injection tool designed with a flexible architecture, built on top of the Bochs and gem5 simulators. [23] gem5 is an inherently slow simulation tool; it is designed for processor architecture research, and executes much less quickly than a dynamic binary translator like QEMU. For its Bochs backend, FAIL* uses an instrumentation system based on a programming language called AspectC++, which allows experiments to listen for particular Bochs events, such as instruction execution and memory access. Unfortunately, this instrumentation takes an already-slow Bochs and makes it slower. Further, FAIL*'s standard fault injection campaigns involve analyzing a complete trace of a test program; during an experiment with running Linux, FAIL* produced multiple gigabytes of trace data before the kernel had even finished decompressing. This combination of factors means that FAIL* is not suitable for continuously injecting faults into a large and complex piece of software like Vivid.

Hailburst resembles FIG-QEMU most closely, in that it uses QEMU and GDB's existing debugging interface for injecting bitflips. It extends the approach by supporting injecting multiple radiation faults within a single execution, and by precisely targeting injected faults to

eliminate any clock-time bias from its fault injections. It is also open source and available at https://github.com/celskeggs/hailburst/.

3. Design of the Vivid Kernel

Vivid provides a framework for building radiation-tolerant applications with redundant multithreading. The resulting applications are tightly coupled to Vivid, which allows the combined system to precisely manage its radiation vulnerabilities. Section 3.1. Redundant Applications on Vivid describes how replicated applications function on Vivid. Section 3.2. Structure of Vivid's Kernel depicts the structure of the Vivid kernel itself. Section 3.3. Vivid API Overview provides an overview of Vivid's memory, communication, and execution abstractions that serve the needs of replicated applications. Section 3.4. Output and State Voting describes how Vivid supports safe redundant multithreading using its voting and scheduling abstractions. Section 3.5. Static Component System describes the static component system that allows engineers to combine applications together with Vivid at compile time to support effective repair and recovery at runtime. Finally, Section 3.6. Safety Properties describes how Vivid establishes its safety guarantees, particularly the guarantees that applications can recover seamlessly from most radiation faults, and that Vivid helps applications prevent serious malfunctions even in the cases where seamless mitigation is not possible.

3.1. Redundant Applications on Vivid

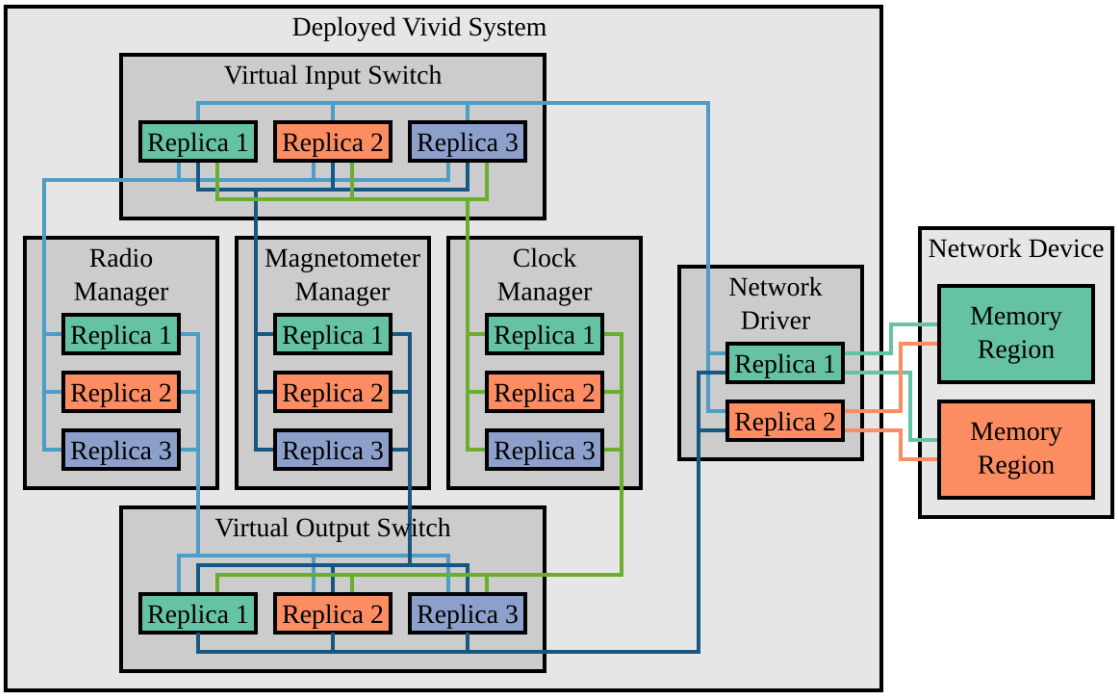


Figure 3-1: A simplified diagram of flight control software implemented using Vivid

Figure 3-1 depicts a simplified design of radiation-tolerant flight software for a spacecraft. This abbreviated design features a single spacecraft bus accessible through a memory-mapped network device, and three device managers (for a radio, a magnetometer, and a mission clock) that communicate with their corresponding devices on the spacecraft bus through a network driver and two virtual network switches. In this example, the flight software engineer implemented the application's functionality by separating out the different parts of the software into six communicating components, which are each replicated individually. The application should be immune to silent data corruptions[7] in its main components, and can usually continue to operate seamlessly in the presence of continuous radiation, but it is possible for rare radiation

[7] Silent data corruptions refer to radiation errors that modify a piece of data without causing a visible crash. An example is an error where a sensor reading is corrupted from the real value to another value that is technically valid, but not the actual value measured. If it were corrupted to an invalid value, by contrast, it would be detectable.

faults to temporarily cause part or all of the application to crash. Vivid will automatically recover from crashes and restore affected parts of the application to operation, but the application may lose some or all of the data it was storing or processing.

An engineer composes a Vivid application from a set of components, some standardized and some unique, and connects them together in a configuration appropriate for the requirements of the spacecraft system. She may define components directly, compose them from smaller components, or both. Vivid will instantiate each component and subcomponent in complete detail at compile time, allocate all of their resources, and predetermine all of their connections. Because applications require minimal initialization at runtime, Vivid can efficiently recover from radiation errors: in many cases, Vivid can repair damage by overwriting the damaged region of memory with the initial version of the region from the boot image. If Vivid did not predetermine the exact resources and interconnections for the damaged component, it would not be able to determine the correct repairs to make, and would need to reinitialize the whole application whenever any portion needed to be repaired.

In the simplified flight software shown in Figure 3-1, each of the application components is composed of three replicas in a redundant configuration, which allows each component to operate seamlessly as long as two of the three replicas are undamaged. The downstream components that receive data from a replicated component will compute a 2-out-of-3 vote over the outputs provided by the replicas, so that they can seamlessly conceal the incorrect behavior of any single transmitting replica. Vivid uses a partition scheduler to ensure that replicas execute in lockstep, so that timing discrepancies will not cause spurious disagreements. Once an error occurs in a component, that component must correct the error quickly, so that it is vanishingly

unlikely that a second radiation fault will occur before the first fault is repaired. Section 3.6. Safety Properties discusses the safety guarantees that are possible under Vivid's voting system.

3.2. Structure of Vivid's Kernel

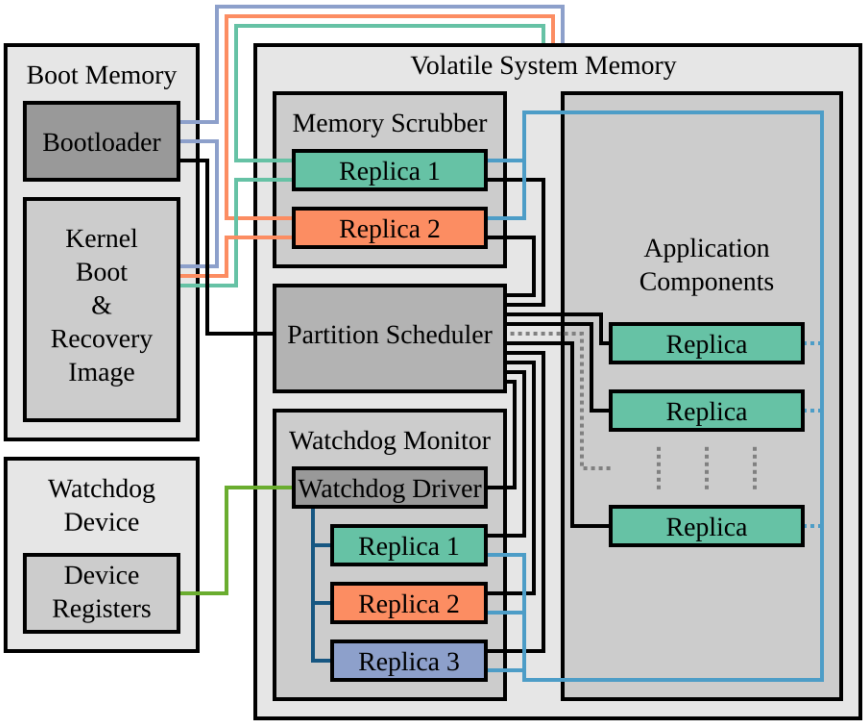


Figure 3-2: A diagram of the kernel components present during runtime.

The components instantiated for a particular application execute alongside Vivid's kernel components. Figure 3-2 depicts the components that implement each of Vivid's responsibilities and how they relate to the application itself. The first responsibility of Vivid's kernel is to maintain the system image in volatile memory, which it does through a bootloader and a memory scrubber. When the system powers up or experiences a reset, the bootloader defined in nonvolatile memory loads the boot image into volatile memory; once the partition scheduler starts running, it executes a memory scrubber that continuously reviews the current contents of memory and ensures that they match the original image. Vivid executes two replicas of the

scrubber, rather than only one, so that they can repair each other. Because Vivid applications are initialized at compile time, rather than runtime, the scrubbers can protect applications' read-only data without explicit registration. However, because the scrubber can repair errors only in read-only memory (including executable code), but not errors in writable memory (such as application state), applications are responsible for defending their own writable memory.

After the bootloader loads the initial kernel image, and Vivid and the application perform any device initialization they require, Vivid proceeds into the partition scheduler. Vivid executes application and kernel code according to a rigid predefined schedule to provide the determinism needed for voting to proceed in lockstep. The schedule specifies the exact order in which replicas and components execute code and exactly how long they execute for; it repeats indefinitely as long as the system is powered on and operational, one scheduling cycle at a time. If a replica cannot complete its execution by its deadline, the partition scheduler forces it to crash and schedules the next component immediately. The partition scheduler will resume crashed replicas from a safe entry point after the memory scrubber has a chance to repair any errors in their code and read-only memory.

In order to ensure that the system remains operational, a physically-isolated and radiation-hardened watchdog device monitors Vivid's execution. As long as the application and operating system agree that the system is operating normally, the watchdog monitor component will feed the watchdog device on a regular basis. An engineer assigns a deadline to each aspect of the system monitored by the watchdog, and sets each deadline to a long enough interval that the components have a chance to recover from any radiation faults they experience. However, if any component fails and does not recover by its deadline, the watchdog monitor will assume that

something has gone irreparably wrong, and ask the watchdog device to force a complete reset of the processor.

Alternatively, if the system is irreparably damaged, and the watchdog monitor cannot request a reset, the watchdog device will notice that it has not heard from the monitor recently enough, and forcibly reset the processor. After it triggers the processor reset, the bootloader will reload the entire application, and the system will reinitialize from a safe state. Because the whole system will be inoperational until the restart is complete, and application data may be lost in the process, Vivid only relies on the watchdog as a repair mechanism of last resort. Although Vivid uses safer and less disruptive repair mechanisms when possible, it still employs a watchdog, because a watchdog device works even in the case that the entire kernel hangs and no repair code at all can run.

3.3. Vivid API Overview

Figure 3-3 provides a brief overview of Vivid's core API, in the form of the built-in components that are available for engineers to instantiate to build their applications. First is the category of memory components, which represent memory regions with different properties. *Scrubbed memory* is immutable, and allows the memory scrubbers to automatically repair any errors in it within one recovery period. This is especially useful for configuration data structures. *General memory* is mutable and not automatically scrubbed. It is useful for replica-local data structures and replica-local data buffers, but requires the individual replicas to take responsibility for error detection and recovery. *Device memory* provides read-only or read-write access to memory-mapped device registers, and is a necessary part of driver components for memory-mapped devices. *Voting notepads* act like general memory, but they automatically

resynchronize their state across a set of replicas by a majority vote on each scheduling cycle, and place less responsibility for error recovery on the application.

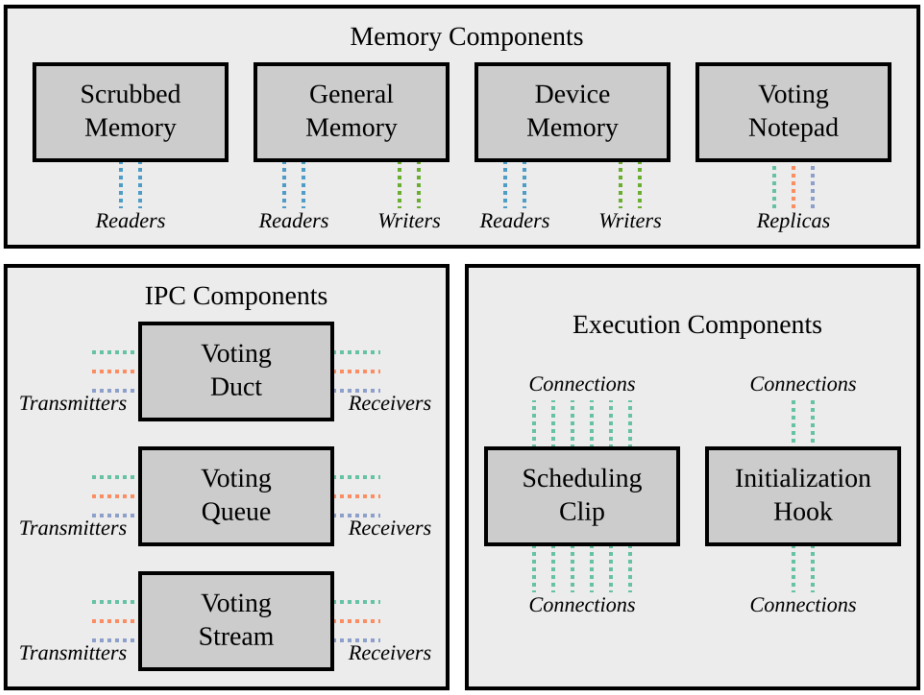


Figure 3-3: An overview of Vivid's core components

The second category is composed of the IPC components. A *voting queue* is a single-reader-single-writer FIFO queue used to reliably communicate between two replicated components, with support for backpressure if the queue fills up. A *voting duct* is a simplified version of a voting queue that does not support backpressure; instead, the receiver must always be ready to immediately process or discard a number of messages equal to the capacity. A *voting stream* is a byte-oriented variant of a voting queue.

The third category is composed of the execution components. Engineers can make code runnable under the partition scheduler by defining an *execution clip* and placing it in the partition schedule. The partition scheduler calls the main function associated with each clip once in each scheduling cycle and allows it to run for up to a configured maximum execution time. To

facilitate device initialization, engineers can also attach code to initialization hooks. Vivid will execute each hook in turn during the boot process and allow each hook to run to completion before starting the partition scheduler.

3.4. Output and State Voting

Vivid supports output and state voting to enable redundant multithreading in applications, but it does not implement any features for this directly in the kernel. Instead, Vivid's IPC library implements the functionality: The IPC library implements voting ducts directly on top of Vivid's shared memory and partition scheduling primitives, and implements voting queues and voting streams in terms of voting ducts. Voting ducts are single-reader single-writer FIFO queues, like voting queues, except that they do not support indicating backpressure; because a duct receiver cannot apply backpressure, it cannot refuse data transmitted over a duct, which eliminates the requirement for bidirectional voting. Ducts will vote on transmitted messages, but not on whether the recipients can handle additional data. Voting queues reintroduce backpressure by augmenting a main voting duct with a backwards voting duct that indicates whether the receiver can accept additional messages. Separating out voting ducts from voting queues has two advantages: first, if a transmitting component knows that a receiving component will always be able to accept data immediately, it does not need to consider what to do in the case that it cannot transmit data, and second, the implementation of voting queues is simpler, because the two votes are cleanly separated from each other.

To ensure that transmitters cannot overwhelm receivers with an unexpectedly large number of messages, engineers define each voting duct with a specific maximum message size and transfer rate. Voting ducts will refuse to accept more messages than allowed within each

scheduling cycle, which allows engineers to design receivers to only handle the maximum transfer rate each scheduling cycle. Because the flow of data to any one of the receiver replicas does not depend on the behavior of the other receiver replicas, a malfunction in one receiver replica will not affect the ability of another replica to receive data.

Engineers must also specify maximum message sizes and transfer rates for voting queues, but the transfer rates are further limited to the number of messages indicated by the receivers through the backpressure duct. Vivid implements voting streams as single-element voting queues that transport all data for each scheduling cycle as a single message.

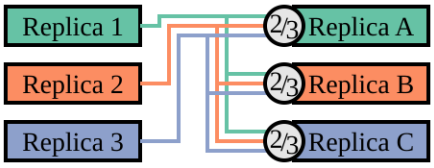


Figure 3-4: Embedded voter architecture.

Voting ducts allow Vivid to vote on component outputs in a distributed rather than centralized fashion, as shown in Figure 3-4, to reduce the number of single points of vulnerability in the system. The engineer defines each application component to have three replicas, and each replica writes its vote of the correct output messages to its own individual buffer each time it executes, using the voting duct API. Each of the downstream replicas that receives data from the upstream component will independently read the messages from each buffer, determine which messages form the majority, and proceed using that majority result as its input. By embedding the voters into the downstream replicas, Vivid avoids the creation of a single voter point of failure: radiation faults that occur during voting will be detected and corrected by the voters that vote on the downstream component's replicated outputs. Because two replicas, if they agree, will overrule any output from a third replica, voting ducts allow Vivid

to silently mask any component output errors, including silent data corruptions, as long as they only affect a single replica's behavior.

Engineers can employ voting ducts (and higher-level voting abstractions) in any arrangement of components, as long as they can design every component to be triply-replicated. Performing each vote three times may make it non-obvious from an exterior point of view which voting outcome is the "true" outcome, but Vivid only needs to establish a single true outcome of a vote when a device driver needs to interact with a single-replicated device; it is otherwise unnecessary.

Voting ducts handle vote splits, such as where the three replicas disagree on the next message (or whether there should be a next message at all), by dropping the data at the offending offset in the message buffer, and allowing the replicas to try again to reach agreement during the next scheduling cycle. Voting queues and streams treat vote splits on the backpressure vote as an indication that the recipients cannot receive any data, and wait until the vote is resolved before continuing to transmit.

A component using backpressure voting needs to account for the possibility that one of its replicas may receive data even when it indicated in the previous scheduling cycle that it could not accept any. This condition can occur when the backpressure vote encounters a 2-1 vote split, with the majority in favor of receiving more data. An engineer must design their components to handle this case, such as by entering the replica into a safe failure state and resynchronizing its mutable state against the other replicas when possible. In particular, she must ensure that backpressure disagreements do not prevent the component from resynchronizing.

3.4.1. Lockstep Scheduling

Vivid executes all components in lockstep using a partition scheduler to synchronize the inputs and outputs transferred between components in the system. Synchronization allows components to compute replicated outputs from matching inputs and state, so that vote splits do not occur simply as a result of natural changes in inputs and state over time. Every replica is responsible for servicing every connected voting duct (or higher-level component) on every scheduling cycle, regardless of whether it transmits or receives any messages that cycle. Servicing includes reading all received data, writing any transmitted data, and updating any internal synchronization state in the voting ducts. As long as each engineer configures every execution clip with a long enough deadline, regular servicing of the voting structures ensures that any possible natural timing discrepancies (such as those caused by microarchitectural processor details) will not affect the correct execution of any of the replicas. An engineer must configure the partition scheduler with a number of clock cycles for each replica's execution clip that accommodates the worst-case code path possible in the clip, so that replicas will only ever overrun their deadlines if they are affected by a radiation error; otherwise, a series of unlucky timing discrepancies could cause a 3-way split vote to occur even in the absence of any radiation faults.

To simplify code path analysis, Vivid only supports implementing device drivers using polling, and does not support device interrupt handlers. Otherwise, a series of interrupts that arrive at an inopportune time might cause timing discrepancies that escalate into a split vote even in the absence of radiation.

3.4.2. State Resynchronization

While voting ducts seamlessly mask radiation errors within a single replica, radiation errors will ordinarily persist inside the replica's state until a repair mechanism corrects them. If an engineer does not implement a mechanism to promptly repair internal errors, there is a danger that a radiation error in one replica of a component might persist for long enough that a second radiation fault could occur in another replica of the same component, leading to a three-way vote split and an externally-visible disruption. Vivid's memory scrubber repairs radiation errors in read-only memory, as long as the application marks the memory appropriately, but engineers of each component with read-write memory are responsible for implementing an appropriate mechanism to correct errors in the mutable state. An engineer might consider the use of voting notepads or *natural convergence* as effective solutions to the problem.

Vivid's support libraries provide voting notepads as a variation on voting ducts. Voting notepads write the state of each replica to a corresponding transmission buffer at the end of each scheduling cycle, and read all replicas' transmissions back at the start of the next scheduling cycle, so that they can compute a majority vote. The voting notepad places the winning result into the next transmission buffer, where the replica can read or write its contents like an ordinary region of memory, and where the replicas can read it again for voting at the start of the next cycle. This approach allows engineers to move part or all of a replica's local storage out of the replica, and extends the seamless masking property of voting ducts to become a seamless correction property of the component's state. Voting notepads will wipe away any error in the protected state by the next scheduling cycle, as long as the error is only present in one replica. It is still possible for a vote split to occur in the presence of unexpectedly high levels of radiation;

in this case, the voting notepad will indicate to the replica code that a split vote has occurred, and the replicas will need to reset their local state to a matching safe value.

Unlike regular ducts, voting notepads store two copies of the transferred state at a time, to ensure that an earlier-executing replica does not overwrite data needed for a later-executing replica to compute the same vote. Voting notepads include an additional set of variables to track which buffer is in use, and they vote on which set of buffers to use for reading and which set of buffers to use for writing before performing the data vote.

Engineers can use *natural convergence* to protect components that need to store large amounts of data for only a short time, especially when the time it would take to vote on the entire state might be too long to allow the use of a voting notepad. An example of such a component would be a telemetry collector component that combines data from many sources and buffers it temporarily until the radio can downlink it from the spacecraft.[8] Natural convergence observes that if one of the three replicas for this component has lost the state for the telemetry buffer, but is still tracking matching metadata for the buffer, the discrepancy between the replicas will only last until they transmit the telemetry buffer. Any new data the replicas receive will match, and the three replicas will quickly converge on a state where they continue to store the newly received data, and do not store the old data, because they have already transmitted it. Any output errors caused by the missing state will only affect a single replica, so Vivid will mask them.

Natural convergence will take longer to correct discrepancies than voting on the entire state of the component, but convergence only needs to occur before the next radiation fault that affects the component, so an engineer may decide that the increased efficiency is the correct tradeoff to make.

[8] This example assumes that a spacecraft does not buffer telemetry in nonvolatile memory, which is often not a realistic assumption, but simplifies this explanation.

When an engineer uses natural convergence, she must take care to ensure that the component's algorithm can only converge to a single possible state; if there are multiple possible states that could be reached, different replicas might converge to different states and fail to reach agreement.

3.4.3. Voting Duct Structure

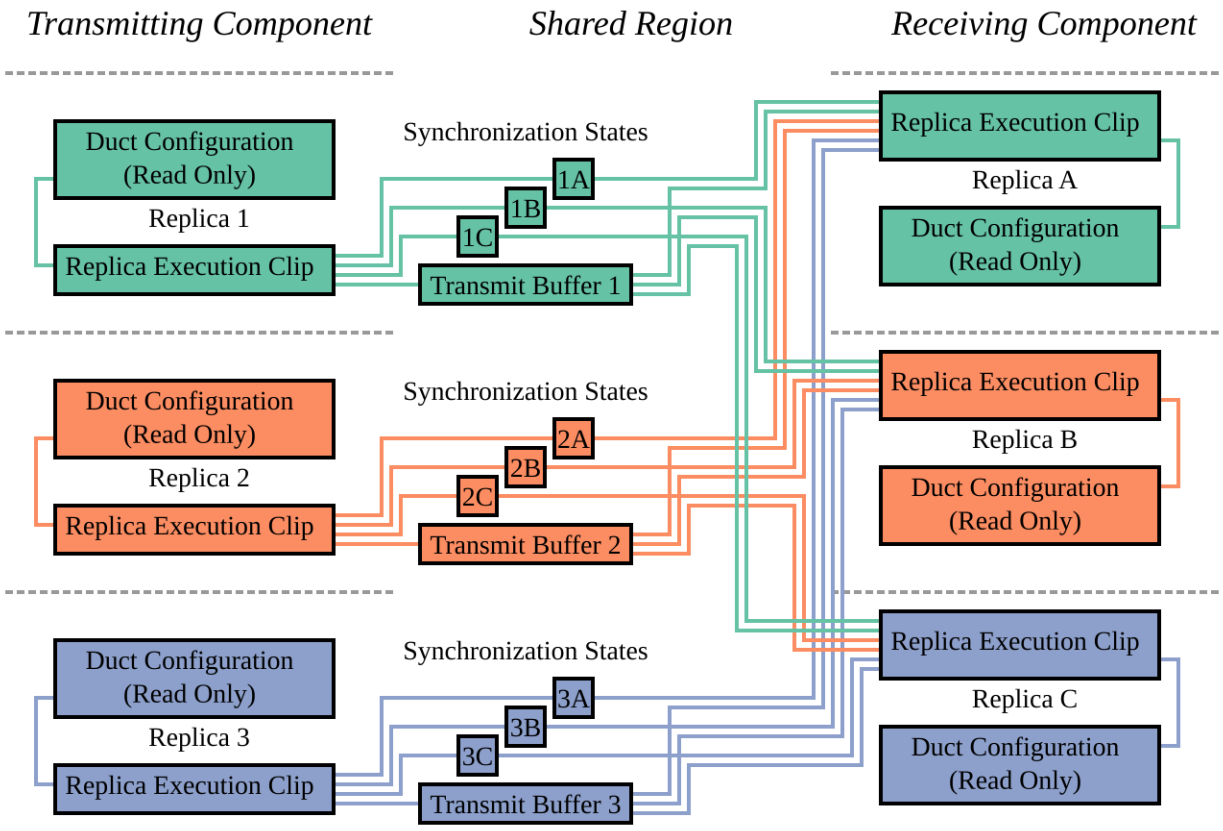


Figure 3-5: Example 3x3 Voting Duct Connectivity Diagram

Figure 3-5 outlines the structure of the voting ducts provided by Vivid's IPC library, using the common example of a duct that connects three upstream replicas to three downstream replicas. The IPC library stores the configuration for the duct itself in read-only memory, with one copy for each replica, and the voting duct API writes each message to a single transmit

buffer corresponding to the transmitting replica. Each upstream replica places metadata about the number of messages it transmitted in three of the nine synchronization state variables at the end of the execution of its scheduling clip; the synchronization variables are cleared by the matching receivers to allow each replica to track which of its communication partners actually executed within the preceding cycle. The receiver clips require the synchronization information to determine which transmit buffers contain data that they should consider during voting, and both the transmitter and receiver clips additionally use the synchronization information to report warnings if their communication partners stop executing.

Each voting duct computes votes on a per-message basis, starting at the first message index and proceeding until either A) it receives the maximum number of messages configured on the duct, B) a majority of transmitter replicas agree that the transmission is complete, or C) it discovers a three-way split vote. Voting ducts *fail disconnected*; once a receiver detects a split vote, it will discard that message and all remaining messages in the current scheduling cycle, as if a fault had caused a temporary disruption in the transmitting component. If a transmitting replica clip exceeds its deadline, it is discarded from the vote.

Because voting ducts only preserve state for a single scheduling cycle, and the state does not persist over longer periods of time, they can recover from errors within a single cycle.

3.4.4. Replicated Components

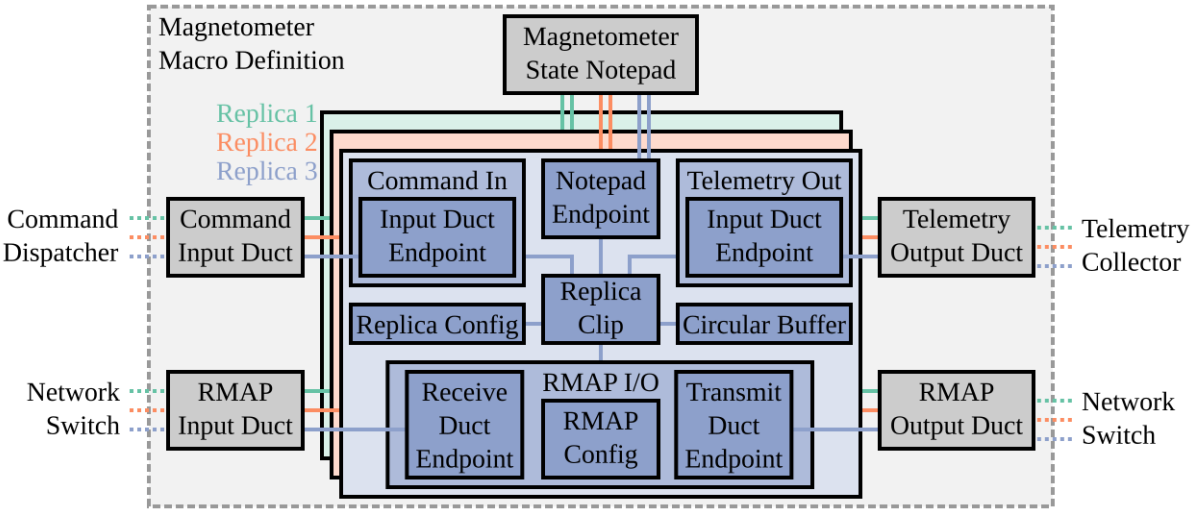


Figure 3-6: Example Replicated Magnetometer Component

Figure-3.6 depicts the structure of an example triple-replicated component that manages a magnetometer instrument. It interfaces with two input ducts and two output ducts: one input duct for commands uplinked from mission control, one output duct for telemetry downlinked back to mission control, and one input and one output duct for it to send and receive packets over the spacecraft bus for communication with the magnetometer instrument itself.

The replicas access the input and output ducts through component wrappers, which encapsulate the communication primitives in higher-level APIs; the component-specific code in the replica clip is not aware of the ducts, and instead uses APIs to receive commands, transmit telemetry, and perform network transactions (using the Remote Memory Access Protocol, or RMAP) to monitor and control the instrument.

The component stores control state about the magnetometer in a voting notepad for synchronization between replicas, and stores static configuration information in a scrubbed memory region. The component also maintains a circular buffer of magnetometer readings within each replica. The voting notepad resynchronizes itself between replicas, and the memory

scrubber protects the static configuration information; the circular buffers naturally converge to empty buffers whenever the component downlinks collected scientific data as telemetry. Because the replicas operate independently, the output ducts mask the existence of any single-replica error, and because all of the state resynchronizes on a regular basis, this component satisfies Vivid's requirements for radiation safety.

3.4.5. Prepare/Commit Drivers

Application components can interact with external devices safely in the same way as they interact with other application components, as long as those devices contain embedded voters that can vote on the messages they receive. Unfortunately, most external devices expect single inputs, not triple inputs, and cannot calculate votes. An engineer can still design a Vivid device driver for a device that does not support voting, but the driver will be more vulnerable to radiation than a regular replicated component. For example, a damaged network driver might stop receiving or transmitting packets, might transmit or receive garbage data, might transmit or receive data from the wrong region of memory (which may be correctly-formatted, but the wrong messages), might transmit or receive more or less data than it should have, or might transmit or receive the same messages multiple times. One option for the engineer to reduce the range and severity of radiation-induced failures is to implement a *prepare/commit driver*, like the network driver shown in Figure 3-7. If the engineer combines a prepare/commit driver with error correction techniques used in ordinary networks, such as checksumming and sequence numbering, she might be able to implement a single network device driver vulnerable only to temporary disruption, rather than more complex data corruption.

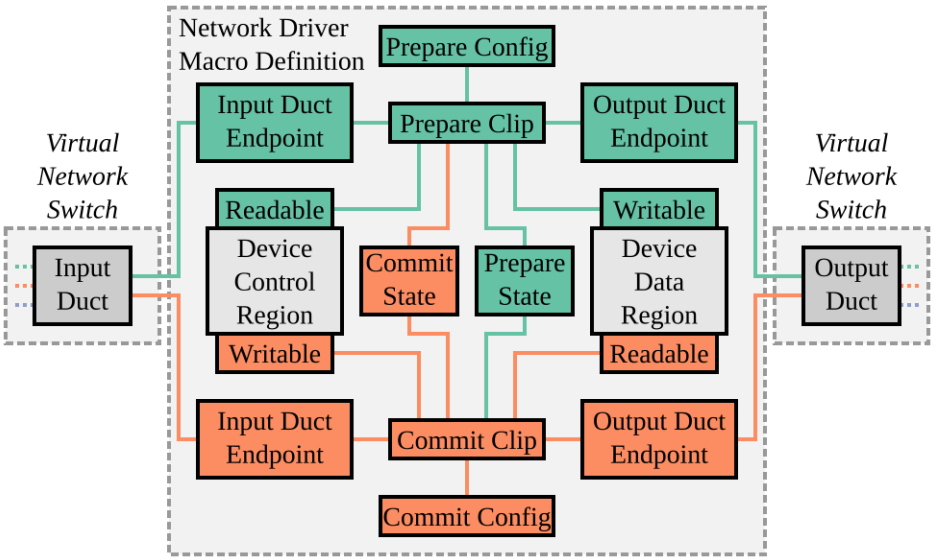


Figure 3-7: Sample Network Device Driver using Prepare/Commit Replication

Figure 3-7 demonstrates the structure of a network device driver using prepare/commit replication, which is appropriate for a network device that contains separate data and control regions. (It is common for network devices to separate data and control regions.) In this example, the network device acts as a link between a physical network interface and a virtual network switch. (The virtual network switch is responsible for routing packets within the flight software.) When the driver receives a packet from the virtual network switch, it writes the packet data to a device data region and triggers the packet's transmission using a write to the device control region. The driver regularly polls the device control region to detect when the device has received a packet from the network, and when it does, it extracts the packet from the device data region and transmits it to the virtual network switch.

The prepare/commit structure separates read and write access to the device control and data regions: the prepare clip reads from the control region and writes to the data region, and the commit clip reads from the data region and writes to the control region. By separating the responsibilities for writing to either region, the design ensures that the device will only transmit a

packet if *both* the prepare and commit replicas agree on it. If one of the replicas malfunctions, the driver's transmissions will generally fall silent, rather than transmit an incorrect message. (There is an exception: depending on the details of the device's memory layout, it may be possible for the commit replica to transmit prepared packets multiple times, instead of once.) Similarly, both replicas must participate in receiving packets from the network, because they will have to both transmit the same message to the voting switch duct to win the vote: the driver's receptions will also fall silent. Neither mechanism can prevent data or length corruptions that occur when data is stored in the device buffers, but an engineer may already need to protect against the possibility of those errors occurring during transmission on the physical communications medium of the network.

3.5. Static Component System

To allow Vivid to fully instantiate the application and kernel components at compile time, engineers implement Vivid components using a combination of a macro system and C's existing static language constructs. For example, an engineer can instantiate a 4 KiB general-purpose memory component by defining a global variable of type `uint8_t[4096]`. To instantiate a read-only memory component protected by the scrubber, she can define a global variable of type `const uint8_t[4096]`. To instantiate a more complex component, like a circular buffer, she would invoke the macro for that component, such as: `CIRC_BUF_REGISTER(buffer_name, sizeof(element), NUMBER_OF_ELEMENTS)`.

To define a new type of application component, an engineer defines a macro parametrized by the instance name, configuration parameters, and connections, and specifies in the macro body the series of global variable definitions (or macro invocations) that instantiate the

parts of the component. Figure 3-8 demonstrates the macro definition process by composing a simplified voting queue component out of two voting duct components. (This definition omits several complexities present in the real voting queue implementation.)

In some cases, an engineer will define additional macros as part of a component definition to access specific static properties of a complex object, such as the list of clips that the partition schedule needs to include for this component, or the list of commands the component can accept from mission control. An engineer may also use C structs to define configuration and state objects associated with components, and may use regular function definitions to provide a runtime API for a component.

```
typedef const struct {
    const char *label;
    duct_t *dataflow;
    duct_t *pressure;
} queue_t;

macro_define(QUEUE_REGISTER, identifier,
             sender_replicas, receiver_replicas,
             max_flow, msg_size) {
    DUCT_REGISTER(symbol_join(identifier, dataflow),
                  sender_replicas, receiver_replicas,
                  max_flow, msg_size);
    DUCT_REGISTER(symbol_join(identifier, pressure),
                  receiver_replicas, sender_replicas,
                  1, sizeof(bool));
    queue_t identifier = {
        .label    = symbol_str(identifier),
        .dataflow = &symbol_join(identifier, dataflow),
        .pressure = &symbol_join(identifier, pressure),
    };
}
```

Figure 3-8: Simplified definition of a voting queue built using two voting ducts

Vivid uses a custom macro language, Siren[9], for most component definitions, rather than using the standard C preprocessor. Siren allows engineers to define multi-line macros more cleanly than they can under the standard C preprocessor, and provides a handful of other useful features, such as compile-time loops. It also allows engineers to define custom macros using Python code, which allows Vivid to perform certain complex operations (see Section 4.1. Debug Log System) at compile time instead of runtime. Because Siren still emits the same intermediate format as the standard C preprocessor, it is compatible with existing GCC-based toolchains. Vivid still uses the C preprocessor as a first stage, before Siren, to handle header inclusion and miscellaneous other preprocessing tasks, such as constant definitions.

3.5.1. Static Memory Management

Because Siren ultimately reduces each instantiation of an application component to a series of global C variable and function definitions, the linker in the standard GCC toolchain can already allocate space for all memory in a Vivid system at compile time. Vivid requires engineers to allocate all memory in this way to eliminate the need for dynamic memory allocation at runtime and simplify the kernel.

Engineers can allocate shared memory (for inter-component communication) in the same way as regular general purpose memory. It becomes shared memory only because a pointer is provided to multiple execution clips. Due to time constraints, Vivid does not yet implement strict memory isolation, but a small amount of work could extend Vivid to define memory access rights at compile time.

[9] Siren's name is a pun referring to a class of dangerous mythical creatures whose voices are famous for luring sailors to jump into the sea. I built Siren so that I could be lured back to C, instead of trying to build an entire operating system in Rust.

Engineers can reference device memory directly by hard-coding a pointer to the physical memory address at which the spacecraft circuitry maps the device memory region. Vivid discourages device drivers from dynamically discovering device addresses and configurations so that it can configure drivers at compile time as much as possible. To ensure that the application does not try to access the wrong memory regions, it is appropriate for device drivers to validate their hardcoded configurations through the standard discovery process; in the case of a mismatch, the best option is to force Vivid to reset the device and driver, and for the driver to refuse to interact with the device and loudly report an error. If the mismatch is caused by a programming error, this approach ensures that the engineer will notice the problem during testing, and if the mismatch is caused by a radiation error, retrying the discovery process after the reset will allow the driver to recover.

3.5.2. Execution Clips

Instead of scheduling threads, with their own registers and stacks that are saved and restored each time they execute, Vivid's partition scheduler schedules *execution clips*[10]. Every time the scheduler executes a clip, it enters the clip's context from the same entrypoint, by calling a main body function specified at compile-time. When the clip's body returns, the scheduler does not call it again until the next scheduling cycle. If the clip doesn't finish executing by the deadline, the scheduler forcibly terminates the execution context and discards the stack and register state. The next time the scheduler executes the clip, it starts the clip again from the regular entrypoint, not from the preemption point. This approach helps applications avoid implicitly sharing data between scheduling cycles: it is not possible for errors in register

[10] The term "clip" comes from the term used in editing video, because an editor may play the same clip as many times as they want, but it will still depict the same things happening. This is in contrast to a standard scheduler task, which may start from a different place and perform a different operation every time the kernel schedules it.

and stack state to go unnoticed for longer than a single clip's execution, because the scheduler will discard the state containing those errors at the end of the scheduling period. (The exception is that uninitialized variables might contain state from other clips or previous executions of the same clip, but properly-implemented C code should not use the values of uninitialized variables.) Similarly, it is not possible for an unexpected delay or infinite loop caused by radiation to persist for more than a single scheduler cycle, because the partition scheduler will terminate the loop at the end of the clip's execution. The strictly deterministic execution sequence defined by the partition schedule also allows engineers to use Vivid as a hard real-time operating system; engineers can precisely analyze the behavior of any schedule to ensure that the flight software can meet its deadlines.

If the scheduler forcibly terminates a clip, either due to a timing overrun or a processor exception (perhaps caused by an access to an invalid memory address or the execution of a badly corrupted instruction), Vivid will wait until at least one of the memory scrubbers completes its next scrubbing cycle before it allows the clip to resume execution. This helps the application avoid repeatedly encountering and reporting local errors in the same replica over and over again, which would flood the debugging output of the system.

Because the partition scheduler never interleaves the execution of multiple clips (and Vivid only runs on a single processor), engineers can implement certain concurrent data structures in simpler ways on Vivid than on other operating systems. Clips do not need to acquire a mutual exclusion lock to read or modify any shared state; if the scheduler preempts the currently-executing clip in the middle of a mutation, the clip will not resume the mutation from the same point; the next clip that accesses the data structure has exclusive access, and when the first clip is scheduled again, it might start the same operation again from the beginning, but not

from the middle. The downside to Vivid's approach is that concurrent data structures become more complex in new ways: instead of needing to protect their state from simultaneous modifications, data structures must support repair of their state after modifications are aborted. If any clip begins mutating a data structure, but never finishes, the next clip to access that data structure will need to roll forward or roll backward the interrupted mutation, or detect that the structure requires repair and wait for the original clip to perform that repair. In practice, the burden of repair falls on the voting infrastructure, and most engineers writing software for Vivid do not need to consider it.

3.5.3. Code Replication Linker

The GCC toolchain ordinarily links the final executable to contain only a single copy of each compiled function and variable, even if the application uses the function or variable in multiple replicas of the same component. Unfortunately, while this deduplication reduces code size, it introduces a single point of vulnerability for replicated components: different components and replicas are not truly independent, so a radiation fault that affects a shared piece of code or data will affect *all* components that use that piece of code or data. Because this type of error can affect all replicas of a component in identical ways, it can lead to crashes and silent data corruption that the application cannot correct with majority voting, because all three replicas might have output that is wrong in the same way.

Vivid selectively reduplicates code by extending the linker to replicate code and read-only data once for each clip, but still deduplicate code within the memory for each clip. Whenever the application defines a clip, Vivid emits a special directive that records the main entrypoint of the clip, and a special linking stage (Section 4.3. Replication Linker) executes the linker an extra time for that particular clip's compiled code, which pulls in all of the functions

and constant variables it needs. Vivid does not duplicate mutable variables; when a clip tries to reference a mutable data structure, it references the single original copy. Vivid links the intermediate objects produced by the extra stages back into the final executable to specify the entry points of the corresponding clips. This mechanism allows application engineers to rely on the guarantee that, if radiation causes code corruption, it will only cause corruption within a single clip, and not in all replicas of the clip.

3.6. Safety Properties

Vivid provides two levels of safety guarantees. *Transparent recovery* covers most radiation faults, and mitigates those faults seamlessly, whether they are crashes or silent data corruptions. *Fallback recovery* covers the remaining range of possible radiation faults; it prevents silent data corruptions from occurring, but may prevent some or all components from executing during repairs, and may reset the state of some components or even the entire system. In both cases, Vivid repairs radiation errors within one *recovery period*, which is a computable duration that corresponds to the maximum amount of time it could take to repair all components in the system, either through the step-by-step repair process described in 3.6.2. Fallback Recovery, or through a system reset and a complete reinitialization of the application. While Vivid guarantees that any radiation faults covered by transparent recovery will not cause any externally-visible misbehavior, fallback recovery does allow for externally-visible misbehavior; however, fallback recovery only allows misbehavior consistent with a specific set of components temporarily ceasing to function, not misbehavior that can only occur as a result of silent data corruption.[11] As an example, a radiation fault that triggers fallback recovery might prevent a

[11] Warnings and errors printed to debugging outputs are not considered externally-visible misbehavior.

system from firing a thruster, but a similar fault could not cause a correctly-designed application to erroneously fire a thruster.[12]

3.6.1. Transparent Recovery

Transparent recovery covers the case where a series of radiation faults occur in fully replicated components (and not in single-replica device driver memory or critical kernel memory) with a spacing of at least one recovery period between each pair of radiation faults. Assuming that every regular application is triply-replicated, and their memory is fully isolated, any single fault will lead to an error in only a single replica of a single component, which the voting infrastructure will seamlessly mask. If the application is properly implemented, the damaged replica will resynchronize its state with the intact replicas either using a voting notepad or through natural convergence. Because Vivid defines the recovery period as the time it would take for all repair mechanisms to engage, then within one recovery period of the original fault, the replicas will resynchronize. Once the system returns to a fully consistent state, it can survive another radiation fault. By repeatedly masking and correcting each subsequent radiation fault, Vivid can seamlessly survive an unlimited number of radiation faults, as long as they affect only the main application and arrive with a sufficient spacing. Because the Vivid kernel and the unreplicatable parts of device drivers are relatively small, compared to the overall size of system memory, it should be reasonable to assume that most faults will only affect the main application.

[12] Specific spacecraft control software can be complex. Engineers must implement applications to operate safely even in the presence of partial failures. For example, if a guidance system depends on receiving data from a gyroscope to decide whether to fire a thruster, it must recognize if the gyroscope software component temporarily fails and fall back to a safe operating mode until the component recovers.

3.6.2. Fallback Recovery

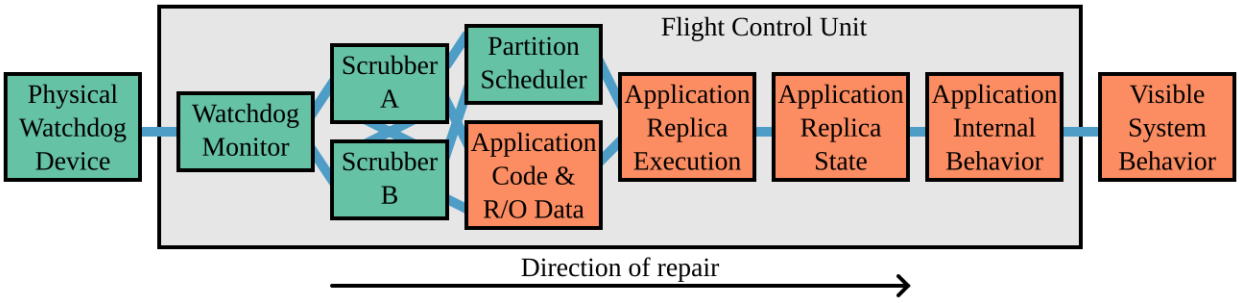


Figure 3-9: Vivid's sequence of repair

Vivid's fallback recovery guarantees, which apply for radiation faults in any region of memory, depend on Vivid's *sequence of repair*, shown in Figure 3-9. Vivid protects each software component using an underlying mechanism that does not depend on that component operating correctly. The ultimate underlying mechanism, which does not depend on *any* software components operating correctly at all, is the physical watchdog device. Because the watchdog device is radiation-hardened and can unilaterally force Vivid to reboot and reinitialize the whole application, it can repair the system even if the entirety of system memory is scrambled and useless.[13] Each subsequent mechanism along the sequence of repair will either operate correctly, in which case it will repair the next mechanism, or will not operate correctly due to corruption, in which case it will be repaired by a previous mechanism. The watchdog device monitors the watchdog monitor clip, which monitors the scrubbers (among other components). The scrubbers will repair each other, the watchdog monitor clip, the partition scheduler, and all code and read-only data in the application.[14] If the partition scheduler is working, and the code and read-only data for the application components are correct, then the application clips will execute

[13] Because Vivid minimizes the complexity of runtime initialization, applications built on it do not take very long to boot, so a system reset does not unreasonably lengthen the recovery period.
[14] The partition scheduler does need to be at least partially operational for the scrubbers and watchdog monitor clip to be functional. If it isn't, then either the scrubbers will be nonfunctional (which the monitor will detect) or the watchdog monitor will be nonfunctional (which the physical watchdog device will detect).

correctly, at least up to the point where they first use invalid replica state. Assuming that engineers have designed the application code to detect invalid states, and either resynchronize or naturally converge back to valid states, proper application execution will lead to valid application states. Once application states become valid, application components will start behaving normally, and the visible behavior of the system will recover.

Beyond guaranteeing that the system will reach a recovered state, Vivid also limits the kinds of misbehavior that may occur during the fallback recovery process. This guarantee depends on the voting architecture: in order for a single application component to visibly misbehave, multiple of its replicas will have to experience radiation faults. In most cases, replica misbehaviors will either lead to a vote split or to a majority of replicas ceasing to operate until repair; in either case, downstream components will receive no messages; it will appear that the damaged component has fallen silent. In order for a downstream component to receive a *corrupted* message, radiation must have damaged multiple upstream replicas in the same way at overlapping times. Because multiple faults affecting multiple replicas are already rare, and most faults lead to crashes, and most faults that don't lead to crashes will lead to a vote split, the probability is vanishingly small that a sequence of radiation faults could cause externally-visible misbehavior, as opposed to behavior consistent with a subset of components ceasing to execute.

Device drivers, as discussed in 3.4.5. Prepare/Commit Drivers, can also fall silent when affected by radiation, but will not introduce data corruptions beyond those possible on the network medium itself. Inside Vivid's kernel, the watchdog and partition scheduler do not directly communicate with the outside world, so a failure in either could lead only to a loss of functionality, not data corruption. A particularly unlucky failure in the scrubbers could lead to the scrubbers overwriting valid memory with garbage contents, but the major follow-on

corruption would likely lead to crashes and vote splits, not undetectable silent data corruption. Therefore, the probability of silent data corruption in a Vivid system becoming visible, even during fallback recovery, is vanishingly small.

3.6.3. Missing Safety Features

The current version of Vivid does not support strict memory isolation between clips, which means that corrupted code can theoretically overwrite data in unrelated regions of memory. The results in Section 9. Evaluation indicate that the system can still tolerate most radiation faults even without this support; future work on strict memory isolation (as discussed in Section 10. Future Work and Summary) will likely improve the safety properties of Vivid beyond those demonstrated in the evaluation.

4. Implementation of the Vivid Kernel

Section 3. Design of the Vivid Kernel describes the high-level structure and approach for Vivid's kernel. This section expands on the kernel's implementation in more detail. Section 4.1. Debug Log System describes the implementation approach for Vivid's debug output system. Section 4.2. Watchdog Device describes how Vivid's custom watchdog device functions. Section 4.3. Replication Linker describes how the replication linker works. Section 4.4. Initialization Hooks and Type Erasure explains the initialization hook system. Section 4.5. Context Switching in the Partition Scheduler provides certain details on how Vivid implements the partition scheduler. Section 4.6. Clip Duration Calibration explains how clip durations are calibrated. Section 4.7. Kernel Loading and Scrubbing describes the bootloader and memory scrubbers. Section 4.8. Build System explains how Vivid's build system works. Finally, Section 4.9. Reliability Configuration lists the configuration options for Vivid's different radiation defenses.

4.1. Debug Log System

The Vivid Kernel and its applications continuously produce debugging information during execution, so that if the system fails (either due to a radiation fault or a bug), an engineer can track down the cause more efficiently. Debugging information varies in importance and output rate, and displaying every line of output directly to engineers during tests would overwhelm them with unnecessary detail. Additionally, string formatting functions, like printf, require significant processing time to execute, because they require Vivid to perform complex string interpolation every time any debug information is generated, which is overhead that Vivid wants to avoid.

Vivid provides a *debug log* system to help address these two challenges. Instead of generating a textual output stream for direct display to engineers, Vivid generates a binary-packed output stream that encodes the arguments associated with each printf-style message, but not the format string of the message itself. For example, executing a logging statement that an engineer specified in `init.c`:

```
debugf(DEBUG, "Calling %u initpoints in stage %u.", count, stage);
```

might result in a message that appears like:

```
[10000.001162002] [          INIT] [DEBUG] Calling 22 initpoints in stage 0.
```

However, the data logged would include only the necessary data, not the whole output:[15]

```
A9 | 60 A4 00 F8 | 12 BB 11 00 00 00 00 00 | 16 00 00 00 | 00 00 00 00 | AF
```

The debug log parser, which is an analysis tool built into SwivelSim (Section 7.6.2. Debug Log Extractor), parses the sequence of binary log entries out of the output file it collects from the flight software and renders them in the human-readable format. It knows where each binary log entry starts and ends, even if preceded by corrupted data, because it searches for the special `A9` and `AF` bytes shown, which delimit the start and end of each debug log entry.[16] The first field within the binary log is always a 4-byte ID for the log message, in this case **F800A460**, which uniquely references[17] the underlying format string, along with other metadata, including the severity (`DEBUG`) and the module (`INIT`, from `init.c`) associated with the original message. The second field, in this case `000000000011BB12` (1162002 in decimal), stores the nanosecond

[15] Separators between binary fields were added for clarity.
[16] If a special byte is needed as part of a debug log body, it is replaced by an **A7** byte followed by the original byte with its high bit flipped. **A9** is replaced by **A7 29**, **AF** is replaced by **A7 2F**, and **A7** is replaced by **A7 27**. These three byte values were chosen to appear infrequently in most data, so that they do not need to be escaped frequently. By ensuring that **A9** and **AF** only appear in output data as delimiters, it is always possible to resynchronize to the correct boundaries for debug logs, even after a period of output corruption in the binary log.
[17] If the ID is not valid, the parser replaces the message with a special "corrupted output" message instead.

timestamp recorded for the debug log.[18] The third and fourth fields, `00000016` (22) and `00000000` (0), are the binary encodings of the arguments passed to the `debugf` function in this case.

In this example, the binary encoding reduced the overall size of the log message from 77 bytes to 22 bytes; if the log message were longer, like many log messages in Vivid and its applications are, the savings would have been even more significant. In addition to reducing the output size, and therefore the amount of time taken to write out the log entry, the binary format also eliminates the need to format any numbers, which can add up to significant performance savings over many debug statements.

In order to implement the binary encoding optimization, Vivid needs to scan the debugf format strings at compile time, extract a complete list of all possible log messages, assign unique identifiers, and generate code for each debug log. The standard C preprocessor cannot accomplish this task, because it does not support string inspection; as an example of why inspection is necessary, consider that Vivid must distinguish between a printf that prints a **`const char *`** with "%s" and a printf that prints a **`const char *`** with "%p": In the former case, it must write out the complete string so that it can be included in the output. In the latter case, it must instead write out the pointer address for the string. To allow code generation based on inspection of string data, Vivid adds an additional debug log extraction stage to the compilation process as a custom Siren macro in Python (see Section 3.5. Static Component System), which parses the format string and generates underlying C code to replace the debugf invocation.

[18] Because Vivid printed this particular debug log before the mission clock was initialized — see 5.1.3. Mission Clock for discussion of the mission clock — it reported the current time as 0.001162002 seconds instead of 10000.001162002 seconds, but the log extractor detected this case and corrected the time, which is why it displayed the rendered time as 10000.001162002.

Figure 4-1 shows an example of the code generated by the Siren macro python code for the previous `debugf` macro invocation, which uses several C tricks to allow format string extraction, unique identifier assignment, and efficient runtime encoding of the resulting debug message.

```
({
  static __attribute__((section ("debugf_messages")))
    const char _msg_format[] = "Calling %u initpoints in stage %u.";
  static __attribute__((section ("debugf_messages")))
    const char _msg_filename[] = "synch/init.c";
  static __attribute__((section ("debugf_messages")))
    const struct debugf_metadata _msg_metadata = {
      .loglevel = DEBUG,
      .stable_id = NULL,
      .format = _msg_format,
      .filename = _msg_filename,
      .line_number = 14,
    };
  struct {
    const struct debugf_metadata *metadata;
    uint64_t timestamp;
    unsigned int arg0;
    unsigned int arg1;
  } __attribute__((packed)) _msg_state = {
    .metadata = &_msg_metadata,
    .timestamp = clock_timestamp_fast(),
    .arg0 = count,
    .arg1 = stage,
  };
  const void *_msg_seqs[] = { &_msg_state };
  size_t _msg_sizes[] = { sizeof(_msg_state) };
  debugf_internal(_msg_seqs, _msg_sizes, 1);
});
```

Figure 4-1: Example implementation code to replace a debugf macro invocation[19]

Instead of diverting the message metadata to a separate artifact during the build process, Vivid includes the metadata in the C binary itself: in Figure 4-1, it defines each metadata field as

[19] No _msg_stable variable is included in this example, so stable_id is set to NULL in the metadata structure. The purpose of the stable_id field is to allow engineers to assign machine-readable stable identifiers to particular debugf messages, such as the BootFromROMKernel message printed on every reboot, so that they can be detected and processed by automated tools — such as the chart renderer described in Section 7.6.3. Chart Renderer — regardless of how the metadata addresses change between compilations of the flight software.

a constant static variable in the custom `debugf_messages` section, which allows the linker to aggregate the metadata into a single section in the kernel executable. The build process strips out the `debugf_messages` section along with the rest of the debug information before deploying the kernel, so the debug metadata does not end up in the version of the system image that executes on the spacecraft's processor. However, Vivid's kernel linker script still allocates space for the `debugf_messages` section in a special region of memory, which means that every message definition in the flight software has a unique address assigned to its `_msg_metadata` variable, such as the address **F800A460** shown before. When the debug log extractor encounters a log entry produced by the code in Figure 4-1, it can look up the metadata address in the original unstripped executable and extract the contents of the `_msg_metadata`, `_msg_format`, and `_msg_filename` fields from the kernel binary, even though Vivid does not include this data on the actual flight control unit.[20]

Vivid directly encodes the structure of the debug log message in C as a packed anonymous struct. This allows it to generate the fields of the debug message in a type-safe manner; as an example, if the engineer tried to pass a string to one of the '%u' arguments in this example, the C compiler would have noticed that the types in the `_msg_state` structure fields did not match the types of the values assigned to those fields, and issued a compilation error. Further, encoding the data as a structure allows Vivid to write all output for all debug messages through a single underlying function (`debugf_internal`) that receives a list of byte regions to include in the final message. The case shown in Figure 4-1 passes only a single byte region, but if the debugf invocation included interpolation of any strings with the '%s' format, Vivid would add those strings as additional data sources to the `_msg_seqs` and `_msg_sizes` array.

[20] The _msg_format and _msg_filename variables explicitly allocate space for their strings, even though this prevents space-saving deduplication, because otherwise C would include these strings in the same region as the rest of the anonymous literal strings in the code, which could not be stripped.

4.2. Watchdog Device

Vivid requires a watchdog to ensure that it can recover even in the presence of inescapable software lock-ups, as described in Section 3.2. Structure of Vivid's Kernel. However, standard watchdog devices used in many ordinary electronics are not fully reliable, and are not appropriate for Vivid. [24] For many devices, software must configure the watchdog with a timeout and explicitly enable it before it can work — which means that runaway software could accidentally write other values to the same registers, disabling the watchdog or extending the timeout uselessly far into the future. Similarly, a runaway loop can repeatedly write data to the watchdog's registers, feeding the watchdog entirely by accident and preventing it from forcing a reset. To ensure that recovery is always possible, Vivid depends on a custom watchdog device that can resist these failure modes.

Table 4-2 shows the registers provided by Vivid's custom watchdog device, which would likely be implemented in a radiation-hardened FPGA on a real spacecraft. The device has no configuration registers; the watchdog's parameters are baked into the circuitry, so software cannot enable or disable the watchdog. The watchdog validates the behavior of the software through three mechanisms. First, reads or writes to unexpected registers will force an immediate reset, because they might indicate that the software is malfunctioning. (They could be the result of an off-target memory copy operation, for example.) Second, the watchdog decides on a period of time when it can be fed, and communicates this through the `early_offset` register. If the watchdog is fed too soon or too late, it will assume the software is malfunctioning and force a reset. Third, the bit pattern that the software must write to the watchdog's feed register to feed the watchdog changes every feeding time; the software must read a recipe from the greet register, compute a predetermined algorithm over the value, and write the resulting food back to the feed

register. If the software writes the wrong value, the watchdog assumes that the software is malfunctioning, and forces a reset.

Address - Register	Behavior on Write	Behavior on Read
`0x090C0000 - GREET`	Force immediate reset.	If time ok: Read recipe. Else: Force immediate reset.
`0x090C0004 - FEED`	If time ok and food matches recipe: Extend deadline. Else: Force immediate reset.	Force immediate reset.
`0x090C0008 - DEADLINE`	Force immediate reset.	Read next deadline.
`0x090C000C - EARLY_OFFSET`	Force immediate reset.	Read early feed offset.

Table 4-2: Watchdog device register map

The arbitrarily-selected algorithm for preparing the watchdog's food is shown in Figure 4-3; any other algorithm that is not trivial to compute (so that it cannot be computed by accident) and that maps most inputs to unique outputs (so that calculating it over the wrong recipe will likely produce the wrong food) would also work, as long as the watchdog device and the software agree on the algorithm.

```
static uint32_t integer_power_truncated(uint32_t base, uint16_t power) {
    uint32_t out = 1;
    for (int i = 15; i >= 0; i--) {
        out *= out;
        if (power & (1 << i)) {
            out *= base;
        }
    }
    return out;
}

static uint32_t wdt_strict_food_from_recipe(uint32_t recipe) {
    // pick out a base and exponent from the recipe and raise the base to that
    // exponent (but make sure the base is odd, because if it's even, it will
    // quickly become 0)
    uint32_t result =
        integer_power_truncated((recipe >> 8) | 1, recipe & 0xFFFF);
    // XOR result by reversed bits of recipe
```

```
    for (int i = 0; i < 32; i++) {
        result ^= ((recipe >> i) & 1) << (31 - i);
    }
    return result;
}
```

Figure 4-3: Vivid's watchdog food preparation algorithm

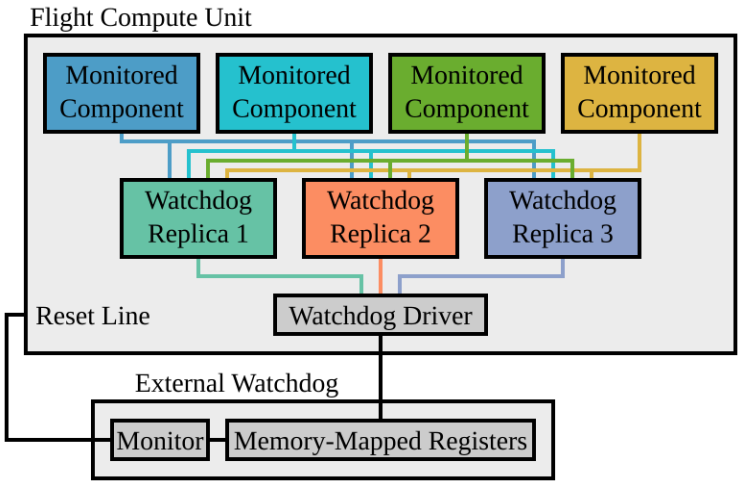


Figure 4-4: Vivid watchdog architecture

To minimize the risk of a malfunction leading to the kernel inadvertently feeding the watchdog, Vivid separates out the watchdog driver into multiple components, as shown in Figure 4-4. Only a single driver clip interacts with the watchdog registers; it detects when it can feed the watchdog, retrieves the recipe from the device, and forwards it to the three watchdog replicas; each replica independently determines whether the monitored components are operating correctly, and only computes the resulting food value if they are. They vote on the resulting food value using a voting duct, and the driver clip receives the result and writes it to the watchdog's registers. If a replica believes that one of the monitored components is malfunctioning and has exceeded its deadline to recover, it will vote for a reset, and the majority votes for a reset, the watchdog driver will intentionally write to a read-only register on the watchdog to force the reset. By splitting up food preparation, such that the watchdog driver clip does not ever run the

food preparation code, Vivid minimizes the possibility that the driver could continue to feed the watchdog after one of the monitored components has malfunctioned.

4.3. Replication Linker

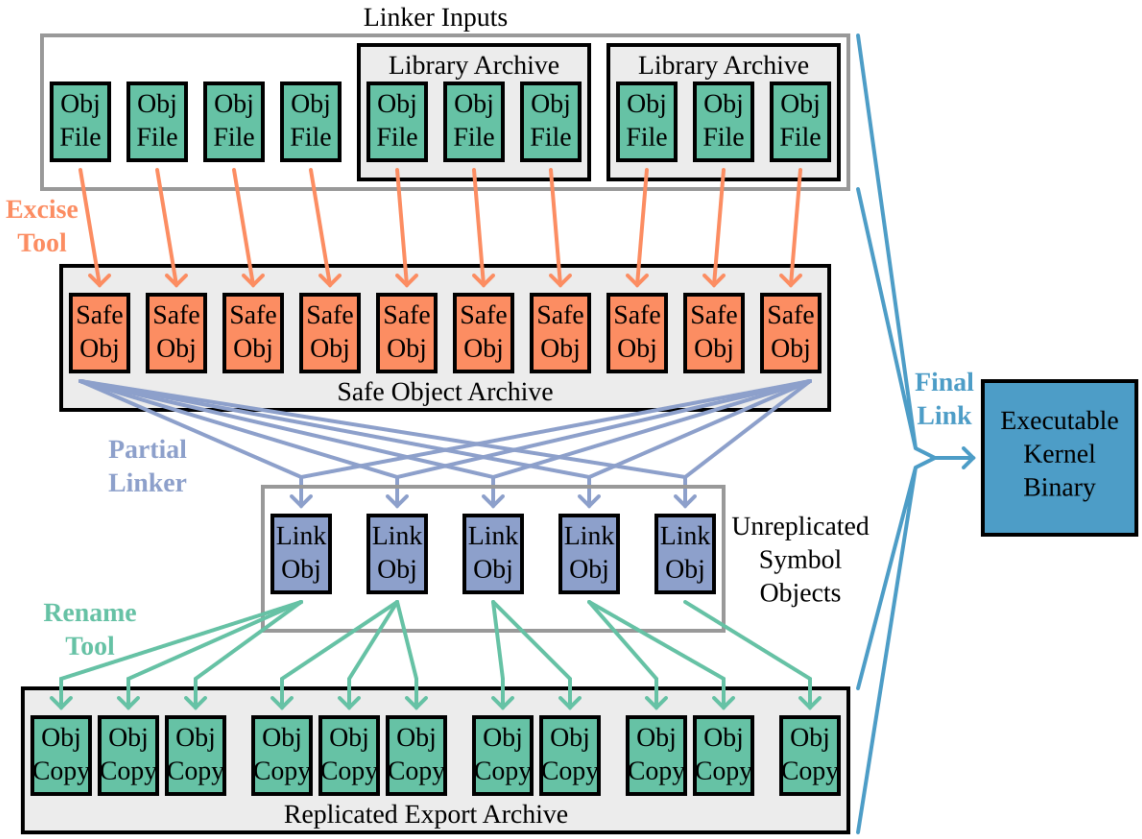


Figure 4-5: Overview of the Replication Linking Process

The replication linker described in Section 3.5.3. Code Replication Linker is a Python script that performs a series of object code transformations according to the replication directives defined by the clip macro invocations. It extracts the list of symbols it needs to define, and the list of original symbols to be replicated to form those symbols, by parsing each input file using `pyelftools` and extracting the list of directives. Based on the extracted list, the Python script executes the transformation sequence shown in Figure 4-5: it excises mutable data out of the

original object files generated by the C compiler to generate “safe” object files, links the safe objects into partially-linked objects for each clip, renames the partially-linked objects according to each of the replicated symbol names, and finally links those replicated objects together with the original object files to form the final kernel executable.

Excision. The excision tool uses the binutils libbfd library to parse the original object files and generate the safe intermediate objects. Like the binutils ‘objcopy’ command (which also uses libbfd), the excision tool reads in the original object file, transforms the symbols and sections one-at-a-time, and writes each of them, along with the metadata, to the newly created output file. As it copies, the excision tool drops any mutable data sections, including .bss and .data, and any special sections (like the initpoints section described in Section 4.4. Initialization Hooks and Type Erasure) that should not be replicated. Each of the dropped sections may contain symbols referenced by relocations defined in other sections of the object file; when the excision tool detects one of these dangling relocations, it will rewrite it to point to an externally-defined symbol, rather than the internally-defined symbol that will no longer exist. As a result, any references to mutable data in replicated code will ordinarily link against the original symbol definitions in the original object files.

If an original symbol was a local symbol (such as a mutable variable defined using the ‘static’ keyword in C), the reference generated by the excision tool will point to a local symbol in the original object file, which cannot be accessed externally. If this condition occurs, the excision tool will reject the object file to prevent a downstream linking error, which will lead to the code in the object file staying unreplicated. To avoid an unexpected lack of replication, Vivid minimizes the use of static mutable variables; engineers will generally want to adjust any static

mutable variables in their source code files to be *global* mutable variables to ensure that the code in those files can be replicated properly.

Partial linking. After the replication linker obtains safe versions of the input object files, it partially links them using the standard binutils linker in '--relocatable' mode. It configures the underlying linker to discard any symbols not used by the replicated function, to minimize the amount of irrelevant code that becomes endlessly copied. Vivid compiles the original object files with '-ffunction-sections' and '-fdata-sections' so that replicating one function or variable within a file does not require replicating all of the others. Vivid specifies a custom linker script for the partial linker to ensure that it only incorporates the .text, .rodata, and debugf_messages sections. (See 4.1. Debug Log System for an explanation of debugf_messages.)

Renaming. The partially-linked object files produced by the linker in relocatable mode define the symbols that should be replicated, but these symbols, along with all other symbols pulled in by the partial link, would conflict with the original definitions in the input object files. The replication linker uses the binutils 'objcopy' command to rename the desired clip entry point symbols to their final names, and convert all other global symbols to local symbols, so that they do not conflict with the original copies.

Once the replication linker renames the partially-linked object files to form the replicated object files, it places them into a library archive. The final executable link is identical to the original link that Vivid would have performed if it didn't use the replication linker, except for the presence of this additional library archive to provide the definitions for the replicated symbols.

4.4. Initialization Hooks and Type Erasure

As mentioned in Section 3.3. Vivid API Overview, Vivid allows applications to register initialization hooks for it to call before entering the partition scheduler. Each initialization hook generates a global variable similar to the one shown in Figure 4-6; because they are defined in the same special initpoints section, the linker places them into a single contiguous array; a linker script provides the address and length of that array to Vivid's startup code, so that it can call each hook during startup. Because the linker does not guarantee a specific order for hooks in the array, each initialization hook specifies a *stage*; Vivid calls all of the hooks in each stage before proceeding to the next.

```
__attribute__((section("initpoints"))) __attribute__((__aligned__(16)))
  const program_init _initpoint_init_internal = {
    .init_stage = STAGE_RAW,
    .init_fn    = (void (*)(void*)) &virtio_device_init_internal,
    .init_param = (void *) &flight_virtio_serial_port,
  };
```

Figure 4-6: Simplified example of an initialization hook

Applications can specify initialization hooks with or without parameters; if a parameter is supplied, it is cast to a `void *`, and the hook function is cast to a `void (*)(void*)` so that it can accept the `void *`. If no parameter is supplied, the same cast is performed, but the parameter is set to `NULL`. This cast-based approach to polymorphism, while common in C language APIs, compromises type safety: the cast will prevent the C compiler from warning if the type of the hook function does not match the type of the parameter.

To reintroduce type safety, Vivid defines the typecheck macros shown in Figure 4-7 using GCC's non-standard builtins; these ensure that, if the type of the initialization hook cannot safely receive the parameter, GCC will report a compile-time error due to trying to cast `void` to `(void`

`(*)(void*))`. Vivid uses Siren's blame_caller macro to ensure that this compile-time error is properly attributed to the user of the `PP_ERASE_TYPE` macro, rather than the macro itself.

```
macro_define(PP_CHECK_TYPE, expr, type) {
    __builtin_choose_expr(
        __builtin_types_compatible_p(typeof(expr), type),
        expr,    /* return expr if type matches */
        (void) 0 /* return void to cause a compile-time error */
    )
}

macro_define(PP_ERASE_TYPE, callback, param) {
    blame_caller { (void (*)(void*)) }
        __builtin_choose_expr(
            __builtin_types_compatible_p(typeof(&callback), void (*)(void)),
            &callback, /* if a function takes void, it can take void* */
            PP_CHECK_TYPE(&callback, void (*)(typeof(*param)*))
        )
}
```

Figure 4-7: Vivid's type erasure API

4.5. Context Switching in the Partition Scheduler

Because Vivid's partition scheduler does not save register state for clips, the assembly code for context switching into a new clip is simple: Vivid changes from IRQ mode to system mode, sets the stack pointer to the clip stack (which is shared across all clips, because stack state is not preserved), clears the floating point status register, and jumps to the clip playback function. The clip playback function validates that the last execution of the clip terminated normally (instead of by timeout or processor abort), and calls the clip's entry point defined by the application. When the application returns from its entry point, Vivid calls the ARM `WFI` instruction and waits out the remainder of the clip's time slice.

When the time slice expires, the processor passes control to the IRQ handler, which discards the clip's stack state, resets the pointer to the top of the partition scheduler's stack space,

and jumps into the partition scheduler (without saving the clip's registers). The partition scheduler selects the next clip to run from the predefined scheduling order, configures the timer to interrupt the processor at the end of the new clip's time slice, and enters the new clip's context.

If an exception occurs, such as a data abort, an undefined instruction exception, or an assertion failure, Vivid does not immediately enter the partition scheduler. Instead, it switches to the exception-handling stack and invokes the exception handler, which will typically record the exception, re-enable interrupts, and call `WFI` to wait out the rest of the time slice, at which point the partition scheduler will take over again. If an exception occurs in the kernel or the exception handler itself, Vivid detects the situation by determining that the aborted code was not running in system mode, and forces an immediate reboot instead of trying to recover.

On reboot, the ARM processor uses an interrupt vector table hardcoded into the Boot ROM, which specifies the bootloader code as the reset vector for the processor. The bootloader provides a minimal exception handler, which Vivid uses during early kernel startup before it can register its own interrupt vector table.

4.6. Clip Duration Calibration

To help engineers calibrate the correct deadlines for each of their applications' clips, Vivid provides a feature to record a high-water mark of the execution duration observed in each clip. It will print out the maximum duration detected for each clip; an engineer can initially define a clip with an unnecessarily long duration and use this feature to measure the actual

execution duration under various conditions.[21] The engineer can take the measured duration, add an error margin, and update the clip's definition to use the new margin.

4.7. Kernel Loading and Scrubbing

Vivid's bootloader and memory scrubbers share a common ELF parsing library that they use to load the kernel into memory and repair the already-loaded kernel, respectively. Vivid embeds the kernel image as part of the bootloader image, and the bootloader passes the pointer to the kernel image region of the ROM to the kernel when it starts, so that the scrubbers know the address of the kernel ELF binary. When the bootloader runs, it loads all of the program segments defined in the kernel; when the scrubbers run, they ignore any segments marked as read-write, so that they do not overwrite any mutable data when they execute.

Because the kernel is too large for the scrubbers to reload its entire image in a single scheduling cycle, and Vivid's scheduler does not save stack or register state for execution clips, the scrubbers must checkpoint and restore their progress in their private memory. Because scrubbing can take a variable amount of time, depending on how much correction is necessary, the amount of data to scrub on each scheduling cycle cannot be computed in advance; instead, the scrubbers monitor the remaining amount of time in their time slices, and when the remaining time drops below a predefined threshold (such as four microseconds), they save their current scrubbing pointers and wait out the remainder of their time slice.

[21] These conditions should include various fault conditions. Voting ducts may take longer to receive messages in the presence of errors, so measuring clip durations in the absence of faults may result in underestimated clip durations that cause unexpected follow-on errors during recovery from previous errors. See section 10.2. Extensions for discussion of this challenge.

4.8. Build System

Vivid uses the SCons build system to compile the kernel and applications, including running additional stages like the Siren preprocessor and the replication linker. [25] SCons tracks completed compilation steps using a database of hashes and a precise list of dependencies, rather than file modification times and partial lists of dependencies, so it only recompiles files that need to be recompiled. As a result, Vivid avoids two common pitfalls that affect software build systems: first, there is no need for an engineer to force a clean recompile at any time, and second, engineers do not need to wait for unnecessary recompilations.

Vivid also uses SCons to build its dependencies, rather than incorporating existing external build systems. In practice, this only covers two libraries: the embedded-artistry libc and zlib. [26] [27] Vivid also links against libgcc, which is distributed with the GCC toolchain and comes precompiled. [28]

4.9. Reliability Configuration

Vivid facilitates evaluating the importance of its different features by providing configuration knobs for different key subsystems, which are listed in Table 4-8. Vivid will still operate correctly with modified configuration settings, but will likely be less reliable.

Configuration Setting	**Effect**
`CONFIG_SYNCH_NOTEPADS_ENABLED (default: TRUE)`	Use voting notepads to synchronize replica states. Otherwise, use unsynchronized memory regions.
`CONFIG_APPLICATION_REPLICAS (default: 3)`	Number of replicas for each application component.
`Options.NoWatchdog (default: false)`	Prevent the watchdog from forcing a reset on watchdog timeout.

`VIVID_SCRUBBER_COPIES (default: 2)`	Number of replicas of the memory scrubber.
`VIVID_RECOVERY_WAIT_FOR_SCRUBBER (default: TRUE)`	Clips that encounter an error and restart will wait for a scrubber cycle to complete before resuming.
`VIVID_REPLICATE_TASK_CODE (default: TRUE)`	Code and read-only data will be linked separately for each clip by the replication linker.
`VIVID_RECOVER_FROM_EXCEPTIONS (default: TRUE)`	When an exception occurs while executing a clip, the clip will restart, and not the entire system.
`VIVID_RECOVER_FROM_ASSERTIONS (default: TRUE)`	When an assertion fails in a clip, the clip will restart, and not the entire system.
`VIVID_CHECK_ASSERTIONS (default: TRUE)`	When an assertion fails, a restart will occur, instead of the failure being ignored.
`VIVID_PARTITION_SCHEDULE_ ENFORCEMENT (default: 2)`	If 0, no preemption. If 1, maximum durations are enforced, but clips can end early. If 2, both maximum and minimum durations apply.
`VIVID_WATCHDOG_MONITOR_ASPECTS (default: TRUE)`	If a monitored component malfunctions for longer than its timeout, the watchdog will force a reset.
`VIVID_PREPARE_COMMIT_VIRTIO_ DRIVER (default: TRUE)`	Each virtio queue driver is split into prepare and commit clips, instead of using a single clip.

Table 4-8: Reliability settings for Vivid

5. The Swivel Spacecraft

Space agencies (such as NASA) do not ordinarily publish complete flight software and simulation environments for real spacecraft. Therefore, in order to evaluate whether Vivid's proposed design can effectively defend flight software against radiation errors, I implemented a representative flight software application on Vivid, which is described in Section 6. The SwivelFSW Flight Software. To create a scenario for the flight software to address, I designed a testbench spacecraft (Section 5.1. Spacecraft Avionics), wrote a list of requirements for the flight software (Section 5.2. Flight Software Requirements), and implemented a simulation for it that would test those requirements (Section 7. The SwivelSim Avionics Simulation).

5.1. Spacecraft Avionics

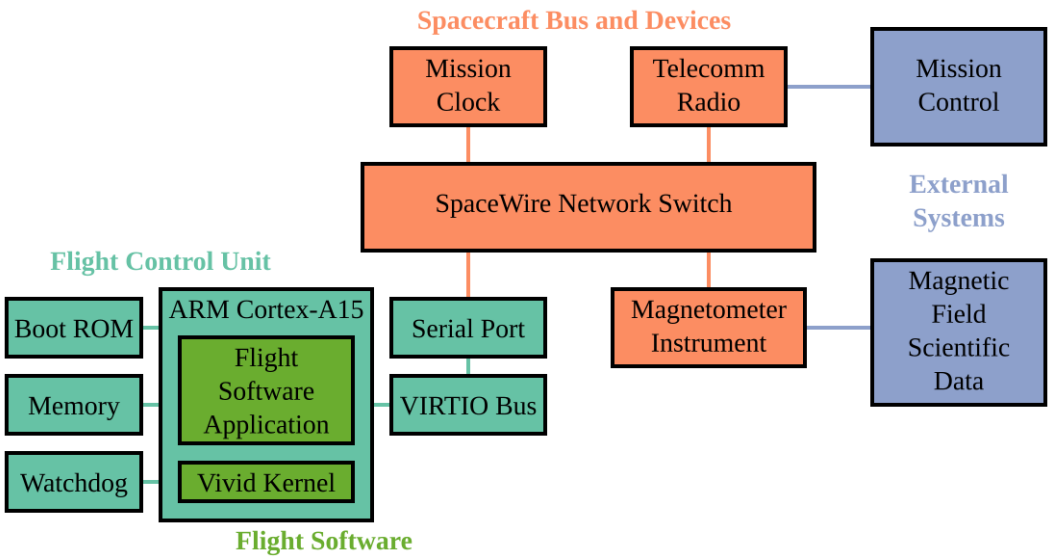


Figure 5-1: Testbench Spacecraft Avionics

Figure 5-1 presents the design of the avionics for the testbench spacecraft. The flight software, including the Vivid kernel, runs on an emulated 500 MHz ARM Cortex-A15 processor,

which is representative of the type of off-the-shelf embedded processors that spacecraft might use. The flight control unit includes 3 MiB of general-purpose memory for system execution and enough ROM to store the boot image. The memory bus maps two I/O peripherals into the processor's physical address space: the watchdog device described in Section 4.2. Watchdog Device and a VIRTIO-based serial port for accessing the spacecraft bus.[22] [29] The spacecraft bus uses a simplified form of the SpaceWire protocol; the flight control unit can send and receive SpaceWire packets over the serial port using the FakeWire protocol defined in Section 5.1.1. FakeWire Protocol. [30] The network switch (Section 5.1.2. SpaceWire Network Switch) acts as the core of the SpaceWire bus, and routes packets between the flight control unit, the mission clock (Section 5.1.3. Mission Clock), the telecommunications radio (Section 5.1.4. Radio), and the magnetometer instrument (Section 5.1.5. Magnetometer). Each of these devices on the spacecraft bus runs the Remote Memory Access Protocol (RMAP) to allow the flight control unit to monitor and control their state through a series of registers. [31]

Mission control communicates with the spacecraft through the radio, according to a command and telemetry protocol defined in Section 5.1.6. Commands and Telemetry.
The testbench design omits several common spacecraft subsystems for simplicity, including thermal regulation, power management, and attitude control.

The simulation of the processor and its memory-mapped devices is implemented using QEMU, and the rest of the avionics are simulated in a separate application; the structure of the simulation is described further in section 7. The SwivelSim Avionics Simulation.

The testbench spacecraft design is significantly simplified compared to most real-world space systems, because implementing a complete simulation and complete flight software for a

[22] VIRTIO is a virtualization-specific interface for virtual machine guests to communicate with virtual machine hosts. VIRTIO does not exist in real hardware, but its memory-mapped I/O interface is representative of the kinds of interfaces available on many real network devices.

truly realistic spacecraft was not possible within the time constraints of this research project. While the testbench spacecraft design demands less from the flight software than a more complete design, the demands it places should be realistic: the flight software must initialize itself and the spacecraft's devices from unknown states, control a radio and a scientific instrument that require constant subsecond management, and process commands and downlink telemetry within strict subsecond time limits. These requirements result in flight software that is similar to real flight software in structure and detail, if not scope, which makes the evaluation of Vivid more accurate to its hypothetical real-world performance.

5.1.1. FakeWire Protocol

SpaceWire ordinarily transmits packets over a complex electrical link layer protocol, which is not easily tunneled over a serial port interface.[23] The testbench spacecraft design includes the FakeWire protocol for reliably tunneling SpaceWire packets across a serial port interface that may experience data corruption. FakeWire defines a selection of data and control characters (see Table 5-2) based on the list of data and control characters allowed in SpaceWire; some control characters are parameterized, in which case they subsume the following four data characters as their parameters. Data characters that have the same serial encoding as a control character are combined with an escape character and shifted out of the control character range, so that control characters are uniquely identifiable in the serial stream. FakeWire combines these characters according to a set of state machines to communicate packets bidirectionally over a serial port interface with flow control.

[23] A serial port interface was the most readily-available way to implement communication in the QEMU-based simulation described in Section 7.2. QEMU Machine Configuration. A more realistic implementation could involve simulating a SpaceWire device that abstracts away the link layer, so that the flight software only needs to send and receive packets.

Serial Encoding	Character
80 (xx xx xx xx)	HandshakePrimary(X)
81 (xx xx xx xx)	HandshakeSecondary(X)
82	StartPacket
83	EndPacket
84	ErrorPacket
85 (xx xx xx xx)	FlowControl(X)
86 (xx xx xx xx)	KeepAlive(X)
[00-7F]	Regular (unescaped) data characters in range 00-7F
[88-FF]	Regular (unescaped) data characters in range 88-FF
87 [90-97]	Escaped data characters in range 80-87
... anything else ...	CodecError

Table 5-2: FakeWire Data and Control Character Encodings

Handshake. The top-level state machine for a FakeWire connection, shown in Figure 5-3, manages the handshake process between the FakeWire exchanges at each end of the connection. Both exchanges start in the `Connecting` state. As long as they have not reached the `Operating` system, then at randomized intervals, they will each send a `HandshakePrimary` message with a random ID to the other and change to the `Handshaking` state. When a FakeWire exchange receives a `HandshakePrimary` message, and has not itself sent one (is not in the `Handshaking` state), it will send back a `HandshakeSecondary` message with a matching ID and move to the `Operating` state. When an exchange in the `Handshaking` state receives a matching `HandshakeSecondary` in response to its `HandshakePrimary`, it also moves to the `Operating` state. By using this protocol to synchronize the exchanges, FakeWire ensures that the exchanges on both ends of an `Operating` connection agree when they have completed a handshake, even

though they operate asynchronously; this avoids the possibility of one end of the exchange sending packets in response to flow control tokens the other exchange had sent in a previous session, and helps avoid letting packets drop as a result of timing errors.

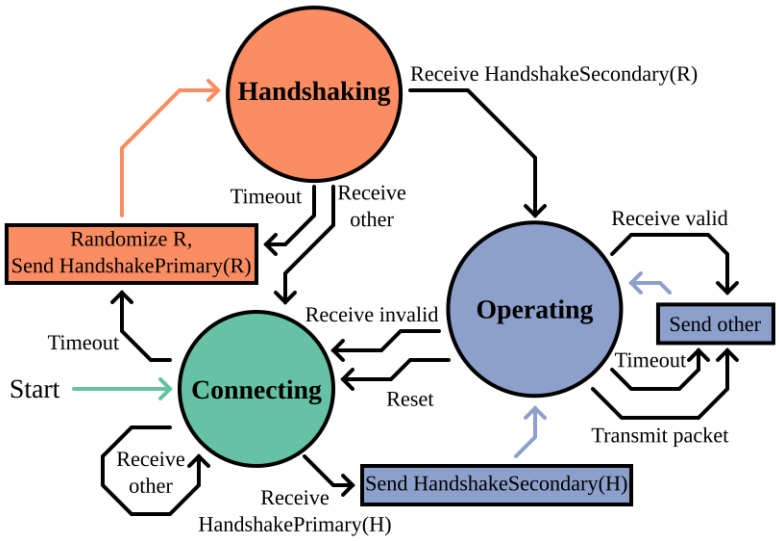


Figure 5-3: FakeWire Handshake Process

Operating mode. When an exchange is in the `Operating` mode, it sends packets in accordance with the series of `FlowControl` tokens the other exchange has sent it, and receives packets in accordance with the `FlowControl` tokens it has previously sent. Each `FlowControl` token indicates readiness to receive one packet of arbitrary size. (Ordinarily an exchange will set a maximum packet size that it supports, and will drop any packets exceeding this size.) The flow control mechanism allows both ends of the link to regulate the packet flow rate in accordance with how many packets of space are available in the receive buffers. FakeWire exchanges transmit `FlowControl` tokens on a regular basis, to remind the other exchange how many packets the exchange is ready to receive, and transmit `KeepAlive` tokens on a regular basis to remind the other exchange how many packets have been sent. If a mismatch occurs, where an invalid token is received, a packet is received when no `FlowControl` token had been sent to

allow it, or a `KeepAlive` token is received that indicates a previous packet had been lost, the FakeWire exchange forces a connection reset.

To transmit a packet, a FakeWire exchange waits until the number of flow control tokens received (the parameter to the most recent `FlowControl` symbol) exceeds the number of packets that it has already transmitted in the current session. Then, it sends a `StartPacket` symbol, the sequence of data characters for the bytes in the packet, and an `EndPacket` symbol. (If the packet is truncated, which is possible under SpaceWire if it was transiting through another link that disconnected, the packet can end early with an `ErrorPacket` symbol instead of an `EndPacket` symbol.) The receiver will unpack the data bytes into its receive buffer, which will necessarily have space for the packet, because it must have previously sent a flow control token in order to allow the packet's transmission.

5.1.2. SpaceWire Network Switch

Connected Device	**Path Address**	**Logical Address**
Flight Control Unit	1	32-39
Telecommunications Radio	2	45
Magnetometer Instrument	3	46
Mission Clock	4	47

Table 5-4: Spacecraft Bus Addresses

The testbench network switch supports SpaceWire's path addressing and logical addressing modes. Table 5-4 shows the addresses for each device. A range of logical addresses (32-39) all route to the flight control unit, to support the optional flight software design (Section 6.4. SpaceWire Virtual Switches) where different software components use different SpaceWire addresses. Otherwise, every device has one path address and one logical address. The switch

silently drops packets sent to any other addresses. It does not support runtime access or reconfiguration of any parameters.

5.1.3. Mission Clock

Address Range: Register	Read Behavior	Write Behavior
`00-03: Magic Number`	Read `0x71CC70CC`	`ErrInvalidAddr`
`04-0B: Clock Time`	Read mission time in nanoseconds.	`ErrInvalidAddr`
`0C-0F: Error Counter`	Read current error counter.	If 0: set error counter to 0. Else: subtract from counter.
`Wrong data length`	`ErrInvalidLength`	`ErrNotAligned`
`Other address`	`ErrInvalidAddr`	`ErrInvalidAddr`
`Corrupted header`	Increment error counter.	Increment error counter.
`Corrupted data`	n/a	`ErrCorruptData`

Table 5-5: Mission Clock Register Map

Table 5-5 lists the network-accessible registers of the mission clock device, which the flight software can use to calibrate the discrepancy between the flight control unit's local clock and the mission clock. The flight software must calibrate the clock time so that the telemetry it sends to mission control uses mission timestamps. In addition to the register for reading the clock time, the mission clock also provides a magic number register so that the flight software can validate that the device is the mission clock (and avoid, for example, sending clock-related messages to the magnetometer if the address is misconfigured), and a counter for how many errors have occurred that could not be reported — such as packets with corrupted headers, which RMAP endpoints cannot respond to.

5.1.4. Radio

Address Range: Register	**Behavior**
`00-03: Magic Number`	Read `0x7E1ECA11`, return `ErrRegisterReadOnly` on write.
`04-07: Memory Base`	Read `0x1000`, return `ErrRegisterReadOnly` on write.
`08-0B: Memory Size`	Read `0x2000`, return `ErrRegisterReadOnly` on write.
`0C-0F: Transmit Pointer`	Pointer in buffer memory for next radio transmission.
`10-13: Transmit Length`	Remaining number of bytes in radio transmission.
`14-17: Transmit State`	`Idle` (0) if not transmitting; set to `Active` (1) to start transmitting buffer data from the Transmit Pointer.
`18-1B: Receive Pointer`	Pointer in buffer memory for next received byte.
`1C-1F: Receive Length`	Remaining number of bytes in receive buffer range.
`20-23: Receive Ptr Alt`	Next Receive Pointer to use after the receive buffer is filled.
`24-27: Receive Len Alt`	Next Receive Length to use after the receive buffer is filled.
`28-2B: Receive State`	`Idle` (0) to drop received transmissions; set to `Listening` (1) to receive the next bytes at the Receive Pointer; `Overflow` (2) if more data is received after both receive buffer and alternate receive buffer are filled.
`2C-2F: Error Counter`	Number of corrupted packets. Write 0 to clear, or any other number to decrement by that number.
`1000-2FFF: Radio Memory`	Access to transmit/receive buffer memory.
`Other address`	`ErrInvalidAddress`
`Corrupted header`	Increment error counter.
`Corrupted data`	Increment error counter, return `ErrPacketCorrupted`

Table 5-6: Radio Register and Memory Map

The radio device allows the flight software to receive and transmit byte-oriented data from mission control. The testbench design includes bandwidth limitations (1 kbyte/second

uplink, 200 kbyte/second downlink), but not any signal latency. The radio exposes RMAP access to twelve registers and an 8 KiB buffer memory range, as shown in Table 5-6; the flight software controls the use of the buffer memory, and can configure transmit and receive pointers to different portions of the buffer as desired. Section 5.1.6. Commands and Telemetry describes the format of the data transferred using the radio interface.

To perform a transmission, the flight software loads the transmission data into buffer memory, sets the transmit pointer and length to that region of the buffer memory, and sets the transmit state to `Active`. As the transmission proceeds, the pointer will increment and the length will decrement, until the length reaches zero, and the transmit state changes back to Idle.

Because transmissions from mission control can occur at any time, the flight software must prepare a receive buffer ahead of time. The flight software must select a region of buffer memory to use for the next receive, configure the receive pointer and length registers to match, and set the receive state to `Listening`. As the radio receives bytes, it will add them to buffer memory, increment the pointer, and decrement the length.

To avoid the possibility that the radio might have to drop data if the receive region runs out, the radio allows the flight software to specify an alternate receive region, which it will automatically switch to when it exhausts the first receive region. Once the flight software detects that the radio has received a transmission, which it can do by regularly polling the pointer and length registers, it must read the received transmission out of buffer memory and, if one of the receive regions is exhausted, provide new pointers for the next receive region.

5.1.5. Magnetometer

Address Range: Register	Behavior
`00-01: Error Counter`	Number of corrupted packets. Write 0 to clear, or any other number to decrement by that number.
`02-03: Power On/Off`	Power state of magnetometer sensors. 0 for unpowered; 1 for powered.
`04-05: Latch On/Off`	Set to 1 to start taking a magnetometer reading, or 0 to cancel. Returns to 0 after reading is taken.
`06-07: X-Axis Reading`	16-bit fixed-point X-axis reading from the magnetometer.
`08-09: Y-Axis Reading`	16-bit fixed-point Y-axis reading from the magnetometer.
`0A-0B: Z-Axis Reading`	16-bit fixed-point Z-axis reading from the magnetometer.
`Other address/read-only`	`ErrInvalidAddr`
`Unaligned write`	`ErrNotAligned`
`Out of range write`	`ErrInvalidValue`
`Corrupted header`	Increment error counter.
`Corrupted data`	`ErrCorruptData`

Table 5-7: Magnetometer Register Map

Table 5-7 presents the registers for the magnetometer instrument. The flight software can power on and off the magnetometer's sensors in response to commands from mission control, and regularly collect magnetometer readings while the magnetometer is powered on. Taking a reading requires turning on the latch register, polling the latch register until it turns back off (which takes around 15 milliseconds). Once the latch turns off, the X, Y, and Z axis registers report the local magnetic field around the spacecraft at the time of the measurement.

5.1.6. Commands and Telemetry

The testbench format for command and telemetry packets, as used by mission control to encode and decode data for uplink or downlink, consists of three layers: first, an encoding for

marking the start and end of command and telemetry packets; second, a generic packet format for command and telemetry packets; and third, a list of body fields for each individual command or telemetry packet type.

Symbol Encoding	**Description**
`00-FE`	Regular data bytes
`FF 11`	Escaped ‘FF’ data byte
`FF 22`	Start of packet marker
`FF 33`	End of packet marker
`FF XX`	Invalid symbol (for `XX` not one of `11`, `22`, `33`)

Table 5-8: Packet Framing Codec

Table 5-8 describes the format of the packet framing encoding: each packet consists of a start of packet marker, a series of (possibly escaped) data bytes, and an end of packet marker. If mission control receives bytes that do not follow this format, it will drop them until it receives the next valid packet. Because the start of packet marker is unique, the testbench design avoids the possibility that a decoder might miss the start of a real telemetry packet because it interpreted it as data for a phantom telemetry packet.

Field Size	**Field Name**	**Description**
4 bytes	Magic Number	If command packet: `0x73133C2C` If telemetry packet: `0x7313DA7A`
4 bytes	Message ID	32-bit unique command type ID or telemetry type ID.
8 bytes	Timestamp	Command or telemetry sender timestamp.
Variable size	Packet Body	Type-specific command or telemetry fields.
4 bytes	CRC	CRC32 checksum of the header and body of the packet.

Table 5-9: Telecomm Packet Format

Table 5-9 describes the structure of the telecommunications packets contained in the framing, which consists of a 16-byte header, a variable-size body, and a 4-byte checksum footer. Mission control will drop any telemetry packets it receives if their magic number does not match the correct value, if its message type ID is not recognized, or if the checksum does not match the packet contents. The meaning of the packet body bytes is determined by the specific command or telemetry type ID specified in the packet header.

Command ID	**Command Name**	**Packet Body Format**
`0x01000001`	`Ping`	4 bytes - Ping ID
`0x02000001`	`MagSetPwrState`	1 byte - Power On (1) or Power Off (0)

Table 5-10: List of Spacecraft Command Formats

Telemetry ID	**Telemetry Name**	**Packet Body Format**
`0x01000001`	`CmdReceived`	8 bytes - Original Command Timestamp 4 bytes - Original Command Type ID
`0x01000002`	`CmdCompleted`	8 bytes - Original Command Timestamp 4 bytes - Original Command Type ID 1 byte - Success (1) or Failure (0)
`0x01000003`	`CmdNotRecognized`	8 bytes - Original Command Timestamp 4 bytes - Original Command Type ID 4 bytes - Original Packet Length
`0x01000004`	`TlmDropped`	4 bytes - Number of Dropped Telemetry Packets
`0x01000005`	`Pong`	4 bytes - Original Ping ID
`0x01000006`	`ClockCalibrated`	8 bytes - Clock Adjustment (signed nanoseconds)
`0x01000007`	`Heartbeat`	[empty packet body]
`0x02000001`	`MagPwrStateChanged`	1 byte - Power Now On (1) or Power Now Off (0)
`0x02000002`	`MagReadingsArray`	8 bytes - Timestamp for Range Start 8 bytes - Timestamp for Range End Repeated for each magnetometer reading: 8 bytes - Timestamp for Reading

		2 bytes - X-Axis Value 2 bytes - Y-Axis Value 2 bytes - Z-Axis Value

Table 5-11: List of Spacecraft Telemetry Formats

Tables 5-10 and 5-11 list the command and telemetry types defined for the testbench spacecraft flight software. The commands include a `Ping` command for confirming that the connection is working (which just transmits a `Pong` message in response) and a `MagSetPwrState` command for enabling or disabling the magnetometer instrument's power and data collection. The telemetry packets include general data handling information (that the mission clock has been calibrated; regular heartbeats; that commands have been received, completed, or dropped; or that telemetry has been dropped because it couldn't be downlinked), responses to specific commands (that a Ping command was received or that the power state of the magnetometer has been changed), and downlinked scientific data (from the magnetometer).

In order to implement the flight software requirements described in 5.2. Flight Software Requirements, the flight software must correctly decode each of the commands, perform the appropriate action in response, and send back the resulting telemetry.

5.2. Flight Software Requirements

The testbench spacecraft flight software addresses 19 requirements, which are validated by SwivelSim (Section 7. The SwivelSim Avionics Simulation) to determine when the flight software is or is not functioning correctly. These requirements are not as formal or well-specified as the requirements defined for real spacecraft flight software, because the testbench design uses them only to guide the automated validation system, not to certify the flight readiness of the

flight software. (Vivid and the testbench flight software are research prototypes, and are not comparable to production flight-quality software.)

The testbench spacecraft defines its requirements in terms of observable behavior, rather than internal flight software state, so that SwivelSim can validate correctness without inspecting the flight control unit's memory. Section 5.1.6. Commands and Telemetry describes the format of the command and telemetry packets referenced in the requirements. Note that this section defines some requirements in terms of more than one "shall" statement if SwivelSim evaluates the statements as a group, rather than separately. (Ordinarily, engineers define requirements in terms of a single "shall" statement.)

This section describes the requirements for the testbench spacecraft in five parts. Section 5.2.1. Command Handling Requirements describes the requirements related to receiving and processing commands uplinked by mission control. Section 5.2.2. Telemetry Handling Requirements describes the requirements related to downlinking telemetry to mission control. Section 5.2.3. Housekeeping Requirements describes housekeeping behavior relating to heartbeats and ping commands. Section 5.2.4. Magnetometer Power Management Requirements describes power management for the magnetometer in response to mission control commands. Finally, Section 5.2.5. Magnetometer Data Collection Requirements describes the requirements related to collecting and downlinking magnetometer readings from the spacecraft.

5.2.1. Command Handling Requirements

ReqReceipt — Within 100 milliseconds after mission control finishes uplinking a spacecraft command, the flight software shall downlink a `CmdReceived` telemetry message confirming the command's receipt.

ReqCmdRecvExpected — The flight software shall not downlink any `CmdReceived` telemetry message if mission control had not uplinked a matching command within the previous 100 milliseconds.

ReqCmdCompleteExpected — The flight software shall not downlink any `CmdCompleted` telemetry messages if mission control had not uplinked a matching command within the previous 300 milliseconds.[24]

ReqCmdSuccess — Whenever the flight software downlinks a `CmdCompleted` telemetry message, it shall indicate that the command succeeded.[25]

5.2.2. Telemetry Handling Requirements

ReqInitClock — Within the first 5 seconds of the simulation, the flight software shall downlink a `ClockCalibrated` telemetry message indicating that it has initialized the mission clock.

ReqNoTelemErrs — The flight software shall downlink only valid telemetry packets.

ReqTelemOrdered — The flight software shall downlink telemetry packets to mission control in monotonically increasing[26] timestamp order.

ReqTelemRecent — When mission control receives a telemetry packet, the packet's timestamp shall indicate a calibrated mission time within the last 20 milliseconds.

[24] The related requirement that flight software indicates command completion is split up into `ReqCompletePing` and `ReqMagSetPwrComplete`.
[25] This requirement is an artifact of the fact that none of the commands required for the testbench spacecraft design involve the possibility of intentional failure; any failure indicates that the flight software is misbehaving.
[26] "Monotonically increasing" means that each timestamp must be equal to or greater than the previous timestamp.

5.2.3. Housekeeping Requirements

ReqCompletePing — Within 200 milliseconds after mission control finishes uplinking a `Ping` command to the spacecraft, the flight software shall downlink a `CmdCompleted` packet indicating command completion.

ReqPingPong — Within 200 milliseconds after mission control finishes uplinking a `Ping` command to the spacecraft, the flight software shall downlink a `Pong` telemetry packet with a matching Ping ID.

ReqHeartbeat — The flight software shall downlink a `Heartbeat` telemetry packet at least once every 150 milliseconds period after it initializes the mission clock.

5.2.4. Magnetometer Power Management Requirements

ReqMagSetPwr — Within 200 milliseconds after mission control finishes uplinking a `MagSetPwrState` command to the spacecraft, the flight software shall have updated the power state of the magnetometer device to the commanded state.

ReqUnchangedPwr — The flight software shall only change the power state of the magnetometer device if mission control uplinked a `MagSetPwrState` command to request the change.

ReqMagSetPwrComplete — Within 300 milliseconds after mission control finishes uplinking a `MagSetPwrState` command to the spacecraft, the flight software shall downlink a `CmdCompleted` telemetry packet indicating command completion.

5.2.5. Magnetometer Data Collection Requirements

ReqCollectMagReadings — While the magnetometer is powered, the flight software shall take a reading from the magnetometer once every 100 milliseconds, with a variance of less than 5 milliseconds.

ReqDownlinkMagReadings — The flight software shall downlink every reading collected by the magnetometer as part of a `MagReadingsArray` telemetry packet within 10 seconds after the reading's collection time; the timestamp shall match the actual collection time to within 500 microseconds, and the three axes of the reading shall match the actual values recorded by the magnetometer.

ReqOrderedMagReadings — The flight software shall downlink magnetometer readings in strictly increasing temporal order, with a minimum of 95ms between subsequent collection times. Each collection time shall fall in the time range specified in the metadata of the `MagReadingsArray` packet, and the time ranges in the metadata of subsequent `MagReadingsArray` packets shall fall in strictly increasing temporal order.

ReqCorrectMagReadings — Each `MagReadingsArray` packet downlinked by the flight software shall contain exactly the set of readings actually collected by the magnetometer during the time range defined in its metadata.

ReqBatchedMagReadings — The flight software shall downlink `MagReadingsArray` no more frequently than once every five seconds.

6. The SwivelFSW Flight Software

SwivelFSW is an implementation of flight software for the Swivel spacecraft (see Section 5. The Swivel Spacecraft) that runs on the Vivid kernel. It illustrates how engineers can implement flight software on Vivid, even though Vivid provides an unfamiliar set of abstractions. The subsystems of Vivid itself were discussed in Section 3. Design of the Vivid Kernel and Section 4. Implementation of the Vivid Kernel; this chapter covers the specific flight software implemented using its abstractions. Section 6.1. Flight Software Overview provides an overview of the structure of SwivelFSW. Each subsequent section describes one of the modules introduced in the overview, and finally 6.12. Schedule Order presents the configuration that SwivelFSW provides for Vivid's partition scheduler.

6.1. Flight Software Overview

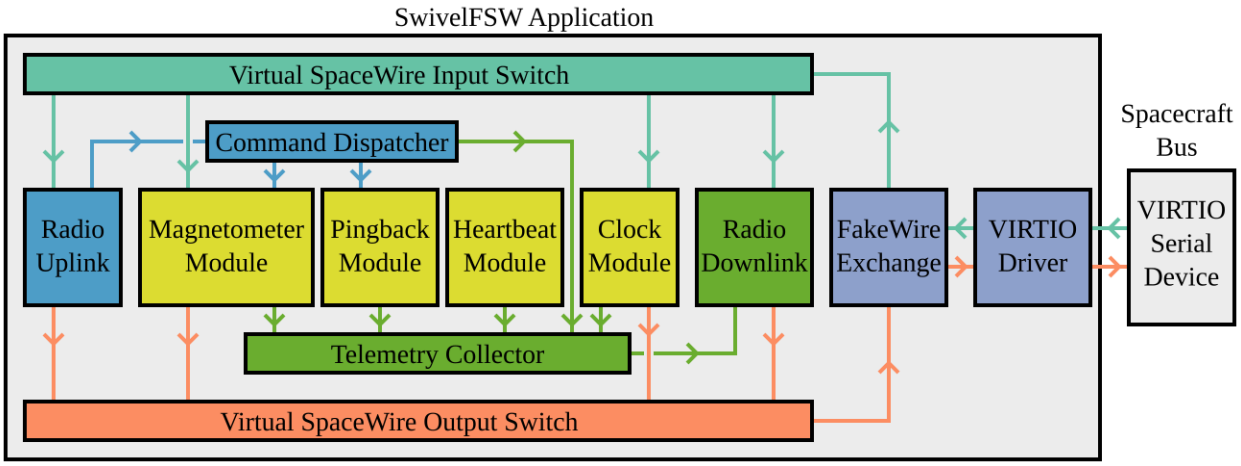


Figure 6-1: Overview for the SwivelFSW Flight Software

Figure 6-1 illustrates the top-level structure of the application components that compose SwivelFSW. All of SwivelFSW's components run with three-replica redundant multithreading, using Vivid's abstractions, except where otherwise mentioned.

While the testbench flight software primarily runs on Vivid, it also supports running on Linux in single-replica mode, replacing the VIRTIO driver with Linux's VIRTIO driver, and eliminating many of the defenses available on Vivid. The current chapter describes the structure of the flight software under Vivid, not under Linux, which is not described in this thesis.

The flight software features four core modules that implement the commands and autonomous behaviors of the spacecraft: a magnetometer module (Section 6.8. Magnetometer Module), a pingback module (Section 6.9. Pingback Module), a heartbeat module (Section 6.10. Heartbeat Module), and a mission clock module (Section 6.11. Mission Clock Module). The core modules receive commands from the radio uplink module (Section 6.5. Radio Uplink and Downlink) through the command dispatcher module (Section 6.6. Command Dispatcher), and transmit telemetry to the radio downlink module (also Section 6.5. Radio Uplink and Downlink) through the telemetry collector module (Section 6.7. Telemetry Collector). Each component that interacts with the spacecraft network bus has its own logical address, and communicates with two virtual SpaceWire network switches (Section 6.4. SpaceWire Virtual Switches) to separately receive and transmit packets. The flight software accesses the spacecraft network bus through a VIRTIO serial port; the VIRTIO driver module (Section 6.2. VIRTIO Driver) interacts with the device itself, and the FakeWire exchange module (Section 6.3. FakeWire Exchange) implements the FakeWire serial protocol described in Section 5.1.1. FakeWire Protocol.

6.2. VIRTIO Driver

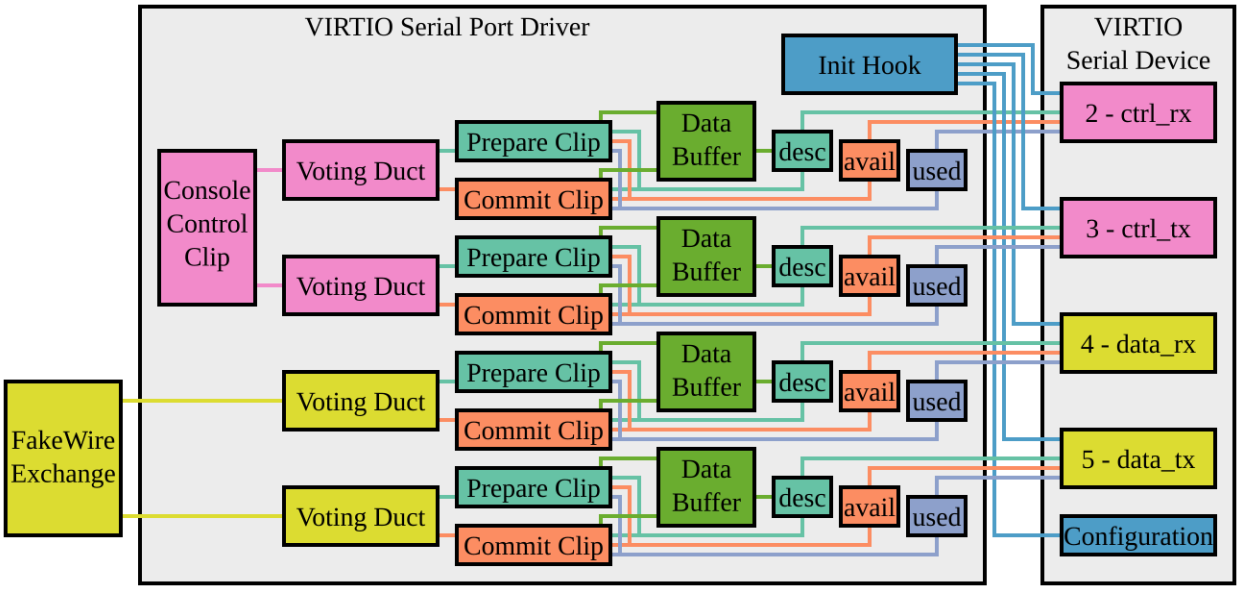


Figure 6-2: VIRTIO Driver Structure

Figure 6-2 details the structure of the driver that allows the FakeWire Exchange component to send and receive data bytes over the VIRTIO serial port. A VIRTIO serial device supports multiple parallel ports, but the testbench design requires only a single port, so the VIRTIO device exposes only port 1. (It cannot expose port 0, because the VIRTIO specification reserves that index for an interactive console.)

In this configuration, the VIRTIO device presents four VIRTIO queues to the driver: two queues to send and receive control messages about the state of the VIRTIO device itself, and two queues to send and receive data bytes across the serial interface. The driver performs basic initialization of the device's data structures through an initialization hook; this step tells the device which memory regions in main system memory contain the descriptor table (`desc`), the available descriptor ring buffer (`avail`), and the used descriptor ring buffer (`used`) for each queue to monitor.

The driver reserves its device memory regions at compile time, along with the memory regions for the send/receive data buffers, using Vivid's static memory allocation components. To minimize runtime scanning, the driver does not enumerate the full list of attached VIRTIO devices to find the serial port device; instead, it hard-codes the known address for the device's memory-mapped I/O region.

The VIRTIO driver services each VIRTIO queue using a separate pair of prepare and commit clips, using the approach described in Section 3.4.5. Prepare/Commit Drivers. The prepare clips are responsible for populating the data buffers and descriptor tables; the commit clips validate the descriptors and place them into the available ring buffer to commit them to the device. Both clips interact with the voting ducts that allow client components to interface with the VIRTIO queues. More information about the descriptors and ring buffers can be found in the VIRTIO specification.

In order to activate the serial port, so that the VIRTIO device will allow the driver to perform data I/O, the driver must send and receive a series of control messages using the control queues. The driver includes a console control clip to track the state of this initialization sequence and perform the activation steps. This clip is triply replicated.

The driver exposes the voting ducts for send and receive data bytes across the serial port, so that the FakeWire exchange component can communicate with the spacecraft bus.

6.3. FakeWire Exchange

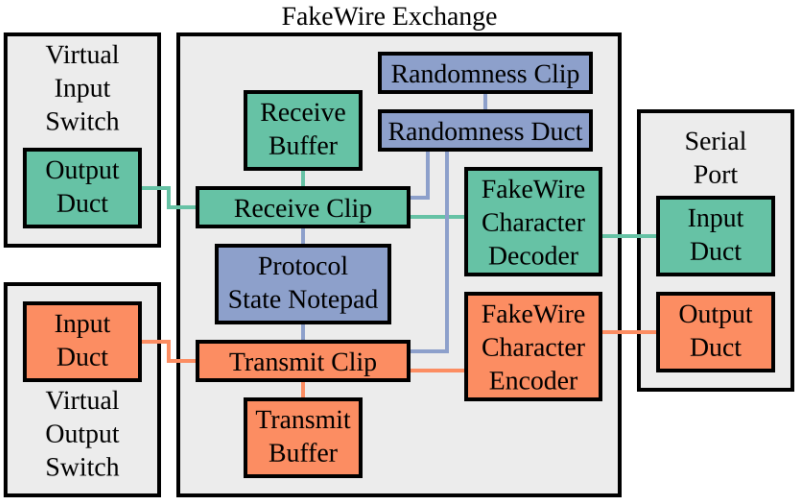


Figure 6-3: FakeWire Exchange Structure

Figure 6-3 describes the structure of the FakeWire exchange component that implements the protocol described in Section 5.1.1. FakeWire Protocol. The exchange splits the FakeWire protocol handling between a receive clip and a transmit clip, which allows them to execute during different parts of the partition schedule. The transmit clip executes after the virtual output switch clip and before the serial port clips, so that, in the common case, it can receive packets from the switch and transmit them on the bus within a single scheduling cycle. The receive clip executes after the serial port clips and before the virtual input switch clip, so that, in the common case, it can receive replies from the bus and send them to the switch for dispatching in the same scheduling cycle. The transmit and receive clips share protocol state through a voting notepad to avoid desynchronization.

If the input switch sends a packet for the exchange to transmit, but the FakeWire protocol is not in the `Operating` state, the exchange will drop the packet. The exchange does not buffer packet data to send in future scheduling cycles; the module that sent the previous request will retry it on the next scheduling cycle, when it does not receive a reply; it is more useful for the

exchange to transmit that new packet, rather than an old packet, and there are not any circumstances in SwivelFSW's design where retransmitting an old packet unchanged would be desirable.

When the receive and transmit clips require random numbers, such as to randomize handshake timeouts to avoid repeated collisions, a singleton randomness clip generates them; by using an unreplicated clip, the three replicas of the exchange can agree on random numbers, which ensures they stay in sync. This does not affect reliability, because the worst case of a failure of the randomness clip is that no random numbers can be generated; the exchange will observe this error, and select non-randomized handshake timeouts repeatedly until the randomness clip recovers.

The encoding and decoding process for FakeWire control and data characters is separated out into separate encoder and decoder components, which wrap the serial port voting ducts and abstract away the details. The code for the encoders and decoders runs in the transmit and receive clips.

6.4. SpaceWire Virtual Switches

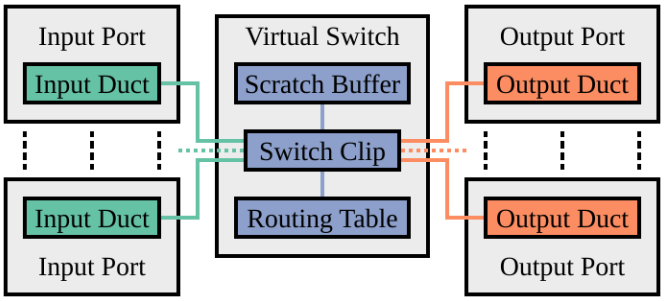


Figure 6-4: Virtual Switch Structure

Figure 6-4 describes the structure of the virtual switch component used in the flight software. This component accepts SpaceWire packets from an arbitrary number of input ports

and routes them to output ports (or drops them) according to the addresses in the packets. If a packet specifies a path address, the switch drops the first address byte and sends it to the matching output port. If a packet specifies a logical address, the switch forwards the packet as-is to the output port associated with that logical address in the switch's routing table. If a packet's address is not found (either the logical address is not in the routing table, or the path address refers to an output port that does not exist on this switch), the switch drops the packet.

The flight software deploys the virtual switch twice: once as an output switch to aggregate packets to transmit over the spacecraft bus (a N:1 configuration), and once as an input switch to dispatch received packets to modules based on their addresses (a 1:N configuration). By splitting up these steps, rather than using a single N:N switch, the flight software can aggregate requests and receive replies within a single scheduling cycle.

Table 6-5 lists the logical and path address assignments used by the switches. Because the switches drop packets to unexpected destinations, the network will neither route a packet that originated from the spacecraft bus back to the spacecraft bus, nor route a packet that originated in one flight software component back to another flight software component.

Destination Device	**Path Address**	**Logical Address**	**Switch**
Physical Bus	1	45-47	Output switch
Radio Uplink Module	2	32	Input switch
Radio Downlink Module	3	33	Input switch
Magnetometer Module	4	34	Input switch
Mission Clock Module	5	35	Input switch

Table 6-5: Virtual Network Switch Address Table

In effect, the combination of the virtual network switches and the physical network switch (Section 5.1.2. SpaceWire Network Switch) produces the SpaceWire network topology shown in Figure 6-6. The physical and virtual portions of the network mirror each other, because there is a mapping between physical components and the virtual components that manage them.

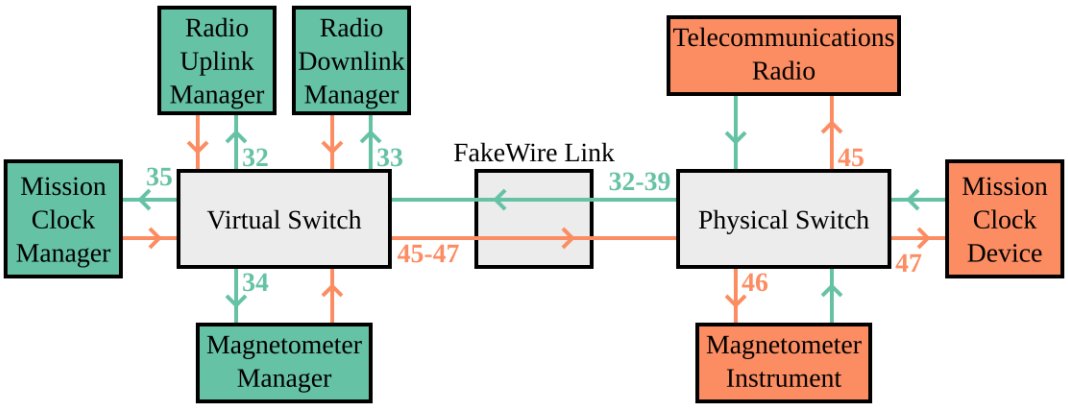


Figure 6-6: Complete SpaceWire Network Map

6.5. Radio Uplink and Downlink

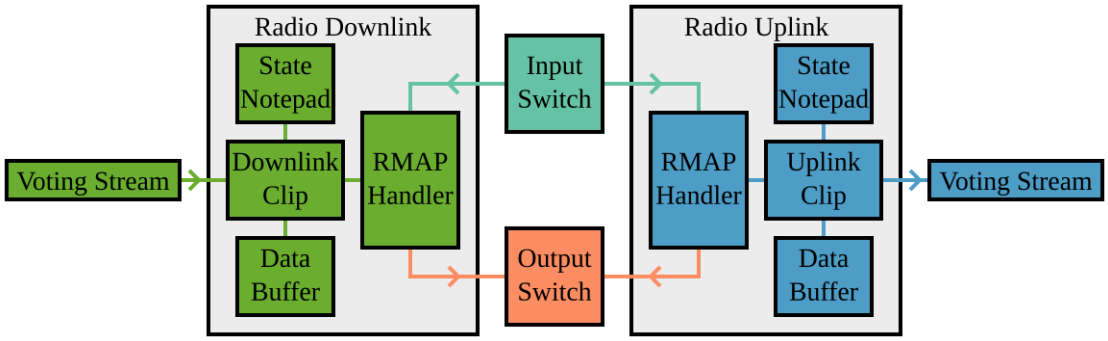


Figure 6-7: Radio Manager Structure

Figure 6-7 shows the structure of the radio uplink and downlink modules, which transfer byte streams of command and telemetry data through the radio device on the spacecraft. The uplink and downlink halves operate separately from each other; each receives a hardcoded half

of the radio's buffer memory (see Section 5.1.4. Radio) and accesses separate registers, so there is no conflict between the two operations.

Access to radio registers and memory takes place through the RMAP protocol (discussed in Section 5.1. Testbench Avionics); the flight software includes a reusable RMAP handler component that encapsulates transmitting RMAP requests in one scheduling cycle and receiving (or not receiving) RMAP replies in the next scheduling cycle. Both radio modules use the RMAP handler APIs to communicate with the radio, which runs within their own clip execution contexts.

Both the uplink and downlink modules synchronize their state in two separate ways: they synchronize their state machines and other control data through voting notepads, and keep unsynchronized buffers for their data. By keeping data in unsynchronized buffers, the modules can avoid voting on pending downlink data every scheduling cycle; however, this lack of voting introduces the possibility that the replicas of one of the modules could diverge. The modules prevent this from happening by using the natural convergence technique (see Section 3.4.2. State Resynchronization): whenever a transmission or reception event completes, the radio buffer values are flushed, so replicas will diverge for at most the handful of scheduling cycles the modules take to complete their current operations.

Because of the complexity of the radio interface, the uplink and downlink modules need to proceed through a multi-step process to perform uplink and downlink operations on the radio. Each of them uses a state machine to track their current steps in this process. Table 6-8 shows the states for the uplink module, and Table 6-9 shows the states for the downlink module. If an RMAP transaction fails, such as due to a bus error, the radio remains in the same state to retry the operation indefinitely. The state machine may advance one or more times each scheduling

cycle, if there is no RMAP transaction or stream I/O to be performed, so in the absence of data to send or receive, both modules tend to idle in a single state: `RAD_UL_QUERY_STATE` for the uplink, and `RAD_DL_WAITING_FOR_STREAM` for the downlink.

Uplink State	**Behavior**
`RAD_UL_INITIAL_STATE` (init)	Entered on reset. Immediately transitions to `RAD_UL_QUERY_COMMON_CONFIG`.
`RAD_UL_QUERY_COMMON_CONFIG`	Queries radio config registers (Magic Number, Memory Base, Memory Size). If configuration matches hardcoded settings, transitions to `RAD_UL_DISABLE_RECEIVE`.
`RAD_UL_DISABLE_RECEIVE` (init)	Sets Receive State register to Idle; then transitions to `RAD_UL_RESET_REGISTERS`.
`RAD_UL_RESET_REGISTERS` (init)	Sets Receive Pointer, Receive Length, Receive Ptr Alt, and Receive Ptr Len registers to zero; then transitions to `RAD_UL_QUERY_STATE`.
`RAD_UL_QUERY_STATE` (loop)	Queries all five receive registers and computes next operations: ● Which radio buffer memory regions to read and forward to the uplink stream. ● Which new values the receive registers should take. Transitions to `RAD_UL_PRIME_READ`.
`RAD_UL_PRIME_READ` (loop)	If there is at least one buffer memory region to read, reads it into the software buffer. On success, or if there is no data to read, transitions to `RAD_UL_FLIPPED_READ`.
`RAD_UL_FLIPPED_READ` (loop)	If there are two buffer memory regions to read, reads the second region into the software buffer. On success, or if there is no more data to read, transitions to `RAD_UL_REFILL_BUFFERS`.
`RAD_UL_REFILL_BUFFERS` (loop)	If there are new values for the receive registers, writes those values back to the radio, then transitions to `RAD_UL_WRITE_TO_STREAM`.
`RAD_UL_WRITE_TO_STREAM` (loop)	If data was collected in the software buffer, transmits it across the uplink stream, then transitions to `RAD_UL_QUERY_STATE`.

Table 6-8: Radio Uplink Module State Machine

Downlink State	Behavior
`RAD_DL_INITIAL_STATE (init)`	Entered on reset. Immediately transitions to `RAD_DL_QUERY_COMMON_CONFIG`.
`RAD_DL_QUERY_COMMON_CONFIG (init)`	Queries radio config registers (Magic Number, Memory Base, Memory Size). If configuration matches hardcoded settings, transitions to `RAD_DL_DISABLE_TRANSMIT`.
`RAD_DL_DISABLE_TRANSMIT (init)`	Sets transmit pointer and transmit length registers to zero, and sets transmit state register to Idle; then transitions to `RAD_DL_WAITING_FOR_STREAM`.
`RAD_DL_WAITING_FOR_STREAM (loop)`	Receives input from downlink stream; if any data received, transitions to `RAD_DL_WRITE_RADIO_MEMORY`.
`RAD_DL_WRITE_RADIO_MEMORY (loop)`	Writes any received input into radio buffer memory, then transitions to `RAD_DL_START_TRANSMIT`.
`RAD_DL_START_TRANSMIT (loop)`	Sets transmit pointer and transmit length to the radio buffer memory region populated with the transmission data, and sets the transmit state register to Active; then transitions to `RAD_DL_MONITOR_TRANSMIT`.
`RAD_DL_MONITOR_TRANSMIT (loop)`	Queries the transmit length and transmit state registers repeatedly until the length drops to zero and the state returns to Idle; then transitions to `RAD_DL_WAITING_FOR_STREAM`.

Table 6-9: Radio Downlink Module State Machine

6.6. Command Dispatcher

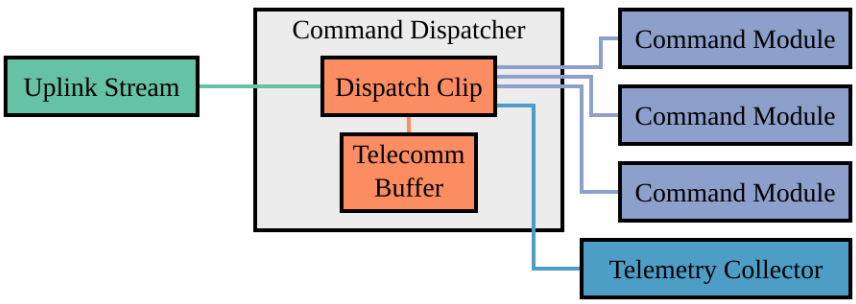


Figure 6-10: Command Dispatcher Structure

Figure 6-10 shows the structure of the command dispatcher, which receives a byte stream from the radio uplink module and decodes it according to the command packet format described in Section 5.1.6. Commands and Telemetry. If the radio has only received part of a packet, the partial packet is stored in a local buffer within each command dispatcher replica until more data becomes available. When the dispatcher receives a full command packet, it records a `CmdReceived` telemetry message, and dispatches the command data to the appropriate module that handles that command using a voting duct; if the command is malformed or unknown (and not handled by any module), the dispatcher records a `CmdNotRecognized` telemetry message and does not dispatch the command.

6.7. Telemetry Collector

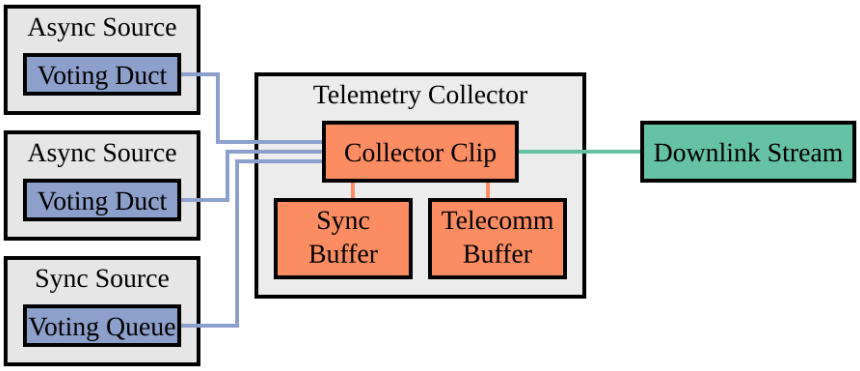


Figure 6-11: Telemetry Collector Structure

Figure 6-11 shows the structure of the telemetry collector module, which receives telemetry messages from multiple sources, encodes them according to the telemetry packet format described in Section 5.1.6. Commands and Telemetry, and transmits them over a byte stream to the radio downlink module. If the radio cannot immediately transmit the telemetry, the collector will buffer it in a local buffer; if the buffer fills up, the collector will start to drop

newly-received telemetry messages. It will count the number of dropped messages, so that it can send a `TlmDropped` message on recovery to indicate how many messages it lost.

When a module requires the telemetry collector to avoid dropping its messages, the telemetry collector allows that module to register as a *synchronous* telemetry source instead of an *asynchronous* telemetry source. A synchronous telemetry source uses a voting queue instead of a voting duct; the collector will store synchronous messages in a separate local buffer, and apply backpressure to the voting queues when they fill up. This allows telemetry sources to make a tradeoff: they can either be able to unconditionally send telemetry messages, which may need to be dropped if the radio falls behind, or can accept that they may be temporarily unable to send telemetry messages, but in return can avoid their messages getting unexpectedly dropped.

6.8. Magnetometer Module

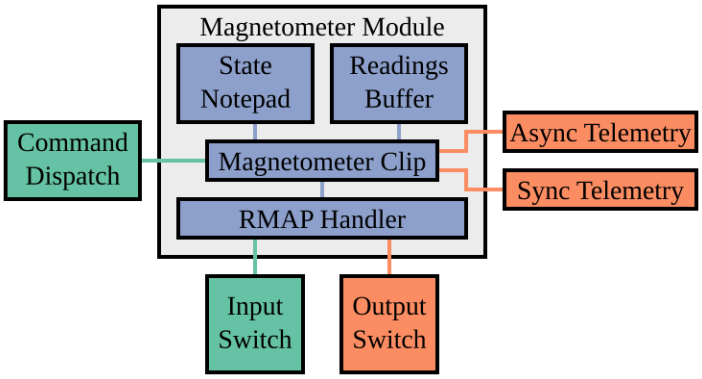


Figure 6-12: Magnetometer Module Structure

Figure 6-12 displays the structure of the magnetometer module, which is responsible for controlling the power state of the magnetometer device in response to commands, and collecting a magnetometer reading every 100 milliseconds that the magnetometer is powered on. The magnetometer module incorporates the same reusable RMAP handler as the radio modules

(Section 6.5. Radio Uplink and Downlink) and uses it to read and write the magnetometer instrument's registers. The magnetometer module places the readings it collects into a local readings buffer, and downlinks them around every 5.5 seconds[27] using the synchronous telemetry endpoint.

If the telemetry endpoint is indicating backpressure, because telemetry is not flowing, the magnetometer continues collecting readings in its buffer until telemetry resumes flowing. If the readings buffer fills up, the magnetometer continues collecting readings, but drops any that exceed the capacity of the buffer. Because readings contain collection timestamps, it will be clear during subsequent processing which intervals of data the magnetometer dropped.

The magnetometer module transmits other telemetry, such as `CmdCompleted` and `MagPwrStateChanged` messages, using an asynchronous telemetry endpoint, so that it does not block on the transmission of these messages.

The magnetometer module tracks which power state has been most recently commanded, and which power state it has most recently successfully applied to the magnetometer; it will retry updating the power state until it is certain that the magnetometer is in the correct commanded state. After reset, the magnetometer module ensures that the magnetometer is powered off.

6.9. Pingback Module

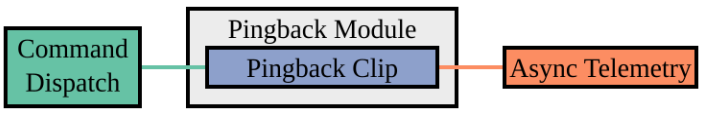


Figure 6-13: Pingback Module Structure

[27] `ReqBatchedMagReadings` requires the flight software to downlink magnetometer readings no more frequently than once every five seconds, and `ReqDownlinkMagReadings` requires the flight software to downlink readings within ten seconds of collection. The magnetometer model uses 5.5 seconds to satisfy both of these bounds with margin for telemetry delays and timing errors.

The pingback module, as shown in Figure 6-13, is straightforward. It receives `Ping` commands from the command dispatcher, and transmits `Pong` and `CmdCompleted` telemetry messages in response. It does not store any state across scheduling cycles.

6.10. Heartbeat Module

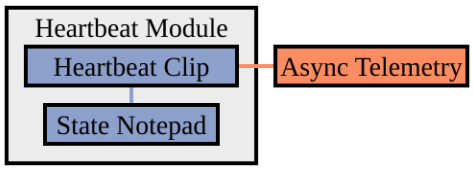


Figure 6-14: Heartbeat Module Structure

The heartbeat module, shown in Figure 6-14, takes no inputs; it keeps track of the last time it sent a heartbeat, and when the current time exceeds that previous time by more than 120 milliseconds[28], it sends another `Heartbeat` message to the telemetry collector.

6.11. Mission Clock Module

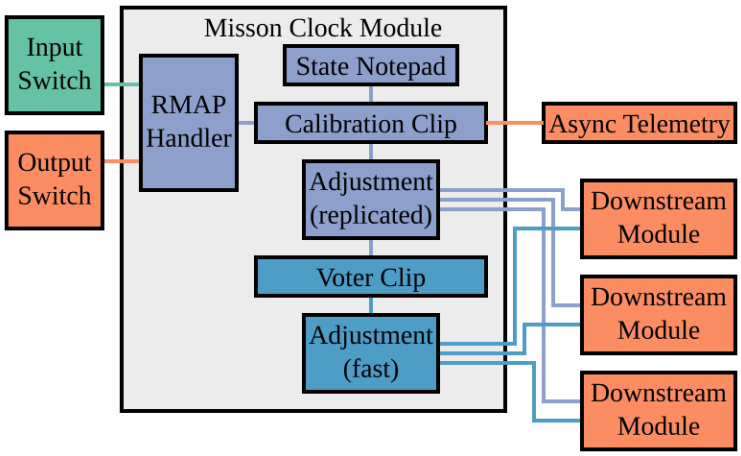


Figure 6-15: Mission Clock Module Structure

[28] `ReqHeartbeat` requires that the flight software downlink `Heartbeat` messages at least once every 150 milliseconds. 120 milliseconds meets this requirement with margin for telemetry delays and timing errors.

The Mission Clock module, shown in Figure 6-15, defines the relationship between local processor time and mission clock time. The clock distributes the *adjustment* it computes to other modules in the system that depend on the current time; this includes modules that report telemetry (which need to specify the current mission time for the telemetry messages) and any modules that report debugging information (which need to include the current mission time in the debug messages). The downstream modules will read the current local time from the ARM `CNTPCT` register and compute the current mission time by adding the adjustment provided by the mission clock. Because almost every component in the flight software needs to compute the current time, the flow of adjustment data does not take place over voting ducts, and was not shown in the overview in Section 6.1. Flight Software Overview.

In order to provide reliable timestamps throughout the flight software without using voting ducts, the mission clock exposes the different adjustment offsets computed by the different mission clock replicas as global variables. The clock provides an API that lets another module compute the majority vote across the three adjustment offsets.

For debug log messages, which are continuously generated throughout Vivid and the flight software, voting is too slow; it would be acceptable for clock adjustments to temporarily deviate (because the requirements in Section 5.2. Flight Software Requirements do not require debug log messages to be correct), so they reference a *fast* adjustment offset available in a global variable. The fast adjustment offset is not replicated, so radiation faults may corrupt it, but a singleton voter clip in the mission clock computes it each scheduling cycle from a majority vote over the replicated adjustment offsets, so the clock normally repairs radiation errors in the fast adjustment offset within one scheduling cycle.

After the mission clock module completes calibration, it ordinarily does not need to recalibrate again while the system runs. However, if a radiation fault damages the value in one of the replicated adjustment offset variables, the module will perform a recalibration against the mission clock device to restore agreement between the replicas.

6.12. Schedule Order

Order	Module	Clip Name	Replicas	Duration
1	FakeWire Exchange	Randomness Clip	1	10 µs
2	FakeWire Exchange	Transmit Clip	3	120 µs
3	VIRTIO Driver	Queue 2 (ctrl_rx) Clip	2 (PC)	25 µs
4	VIRTIO Driver	Console Control Clip	3	15 µs
5	VIRTIO Driver	Queue 3 (ctrl_tx) Clip	2 (PC)	10 µs
6	VIRTIO Driver	Queue 5 (data_tx) Clip	2 (PC)	70 µs
7	FakeWire Exchange	Randomness Clip	1	10 µs
8	VIRTIO Driver	Queue 4 (data_rx) Clip	2 (PC)	25 µs
9	FakeWire Exchange	Receive Clip	3	100 µs
10	Input Switch	Switch Clip	3	60 µs
11	Radio	Uplink Clip	3	40 µs
12	Command	Dispatch Clip	3	100 µs
13	Magnetometer	Magnetometer Clip	3	110 µs
14	Mission Clock	Calibration Clip	3	10 µs
15	Mission Clock	Voter Clip	1	10 µs
16	Pingback	Pingback Clip	3	10 µs
17	Heartbeat	Heartbeat Clip	3	10 µs
18	Telemetry	Collector Clip	3	150 µs

19	Radio	Downlink Clip	3	70 µs
20	Output Switch	Switch Clip	3	60 µs
21	Vivid Kernel	Memory Scrubber Clip	2	100 µs
22	Vivid Kernel	Watchdog Voter Clip	3	30 µs
23	Vivid Kernel	Watchdog Monitor Clip	1	10 µs
		TOTAL	56	3155 µs

Table 6-16: Partition Scheduler Configuration for the Flight Software
(Durations listed for a single replica. "(PC)" indicates that the two clips are prepare/commit replicas.)

Table 6-16 displays the partition scheduler configuration for the flight software, with replicated copies of the same clip collapsed into a single row. SwivelFSW schedules its components in an order that matches the data flow order: this allows data to flow in from the spacecraft bus, to be processed, and to flow out within a single scheduling cycle. As an example of the alternative, consider the case where input switch clip (10) and the output switch clip (20) were swapped, so that the output switch clip had index 10 and the input switch clip had index 20. If the magnetometer clip (13) wanted to send an RMAP packet to the spacecraft bus and receive a response, it would take one scheduling cycle to get back around to the output switch (13 to 10), another scheduling cycle to get to the transmission point, get transmitted, receive a response, and pass the response to the input switch (10 to 2 to 9 to 20), and then another scheduling cycle to get back around to the magnetometer clip (20 to 13). The order actually selected allows more efficient data flow, not just for the flow of RMAP packets, but also for the flow of commands and telemetry, to minimize the latency of the telecommunications system.

7. The SwivelSim Avionics Simulation

SwivelSim is a fully deterministic avionics simulation for the Swivel spacecraft described in Section 5. The Swivel Spacecraft. It includes simulations for each component of the spacecraft design, including the flight control unit, the devices on the spacecraft bus, mission control, and the magnetic field around the spacecraft. It automatically generates inputs for the spacecraft, such as commands for mission control to uplink, and validates the resulting spacecraft behavior against the requirements listed in Section 5.2. Flight Software Requirements.

SwivelSim integrates with Hailburst, described in Section 8. The Hailburst Fault Injection System, to provide radiation fault injection for the emulated processor. Section 7.1. Simulation Overview provides an overview of SwivelSim's structure, and the remaining sections describe the different parts of the simulation in detail: Section 7.2. QEMU Machine Configuration describes how SwivelSim simulates the flight software on the flight control unit, Section 7.3. Timesync Protocol describes the timesync protocol that synchronizes the different processes in the simulation, Section 7.4. Watchdog Device describes the simulation of Vivid's custom watchdog device, and Section 7.5. Spacecraft Bus Simulation describes the simulation for the rest of Swivel. Finally, Section 7.6. Analysis Tools presents the tools available to analyze SwivelSim experiments after they are executed.

7.1. Simulation Overview

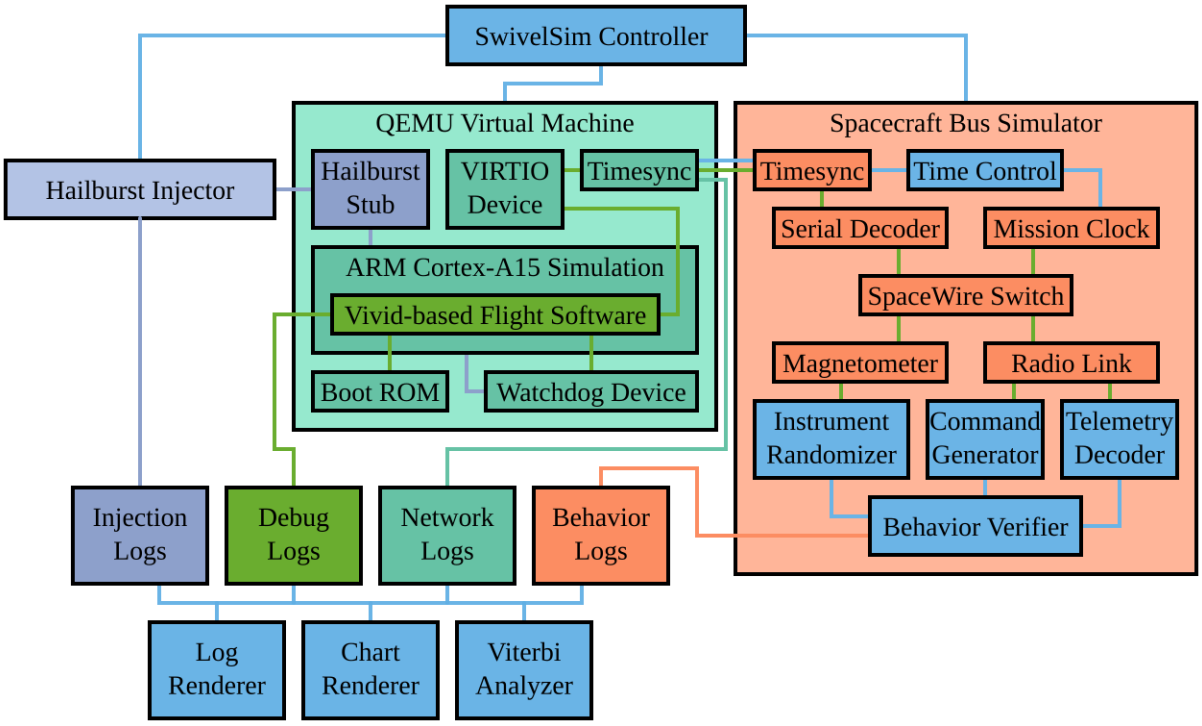


Figure 7-1: SwivelSim Simulation Overview

Figure 7-1 provides an overview of SwivelSim. There are three major components of the simulation that execute during each testing trial, under the supervision of SwivelSim's experiment controller: the Hailburst process that injects faults, the QEMU process that emulates the ARM processor and its memory-mapped peripherals, and the spacecraft bus simulator that models the rest of the spacecraft and monitors the behavior of the flight software. These components generate log files during execution that describe the inputs the simulation provided and the behavior the simulation observed. The three components execute in lockstep through simulated time to ensure that the system is fully deterministic; when one component is executing, the other two are waiting for it before they progress forward. SwivelSim provides multiple tools, including a debug and network log renderer, an execution chart renderer, and a Viterbi-based

reliability analyzer, for an engineer to use to process and understand the logs produced by a simulation.

7.2. QEMU Machine Configuration

SwivelSim uses QEMU to emulate the Cortex-A15 ARM processor for SwivelFSW to run on. It configures QEMU using a modified version of the standard ARM `virt` machine type, which is simpler than most real-world ARM platforms. This machine type includes support for memory-mapped VIRTIO devices, which are devices optimized for efficient virtualization. [29] SwivelSim configures a VIRTIO serial device to allow the flight software to communicate with the spacecraft bus using the FakeWire protocol described in Section 5.1.1. FakeWire Protocol.[29] SwivelSim connects the serial device to an underlying character device connected to the spacecraft bus simulator process.

Ordinarily, QEMU provides the emulated ARM machine with the real current clock time from outside of the simulation. While this makes sense for a machine that a user directly interacts with, it results in a nondeterministic simulation; small variations in host load can result in variations in exactly how many instructions QEMU executes within a period of time, and the simulation will be sensitive to minor scheduling variations of the host scheduler, which will obscure any hard-real-time properties of the simulated system.

To ensure that QEMU executes deterministically, SwivelSim places QEMU into icount (instruction count) mode, which advances QEMU's simulated clock time in lockstep with instruction execution. Normally, QEMU dynamically recompiles blocks of guest instructions into blocks of equivalent host instructions to execute code efficiently, which means that it executes an

[29] The VIRTIO serial device is one of the highest-efficiency serial devices that can be simulated under QEMU; most or all other serial devices require the guest to transmit and receive a single byte at a time. SwivelSim uses a serial device because QEMU does not emulate any real SpaceWire devices.

entire translation block of instructions before it checks for interruption. However, in icount mode, if executing a full translation block would advance the current time past the next timer, QEMU will generate a new shorter translation block that only executes for exactly the number of instructions remaining until the next timer.[30] Because the devices emulated by QEMU will use the simulated clock time instead of the real clock time, the flight software will see consistent timings for the operations it performs.

7.3. Timesync Protocol

SwivelSim needs to connect the QEMU's emulated VIRTIO serial device to the external spacecraft bus simulator, but QEMU's standard mechanisms for connecting a character device to an external process are limited to asynchronous communication. Once a message is sent, QEMU will continue executing the guest until it receives a message back, and since messages between QEMU and the spacecraft bus simulator may take varying amounts of time depending on details of the host system, such as current CPU load, every simulation trial using asynchronous communication would vary in small ways based on exactly when the host kernel delivered each message. Over time, these small variations would accumulate, and the behavior of the simulation would diverge. To ensure determinism, SwivelSim patches QEMU to pause whenever the flight control unit sends a message to the spacecraft bus and wait until the bus simulator returns a response before proceeding. This mechanism is provided by SwivelSim's *timesync protocol*, which implements a custom character device backend for QEMU that SwivelSim can connect directly to the VIRTIO serial port.

[30] There is a small deviation based on the length of time it takes to execute each instruction. For example, in 500 MHz icount mode, each instruction takes 2 ns; therefore, QEMU may need to execute 1 ns past the next timer to align with the next multiple of 2 ns. This is because QEMU does not support partial execution of instructions.

The timesync protocol uses a unix domain socket to communicate between the QEMU emulator and the spacecraft bus simulator. Whenever serial data bytes are sent from the flight software, they are stamped with the transmission time; QEMU will pause and wait until it receives a reply from the bus simulator, which will execute its simulation until it reaches the specified simulation time, receive the bytes in question, and either send back response bytes immediately or indicate that no bytes are currently available. The bus simulator can request that QEMU register a timer for a specific future simulation time, when the spacecraft bus might have additional serial bytes to send to the flight software, even if the flight software hasn't sent anything else by this point. As a result, the spacecraft bus simulator operates like a module of QEMU, allowing SwivelSim to run experiments completely deterministically.

7.4. Watchdog Device

SwivelSim implements Vivid's custom watchdog as part of the QEMU process, which allows it to operate independently of the main spacecraft bus. QEMU provides several implementations of standard off-the-shelf watchdog timers, but none of them are sufficiently similar to the watchdog device design described in Section 4.2. Watchdog Device. Therefore, SwivelSim implements Vivid's custom watchdog design as a patch to QEMU and statically enables it in the machine configuration. When the watchdog decides that it needs to force a reset, it uses QEMU's existing watchdog interface to perform a reset of all the components simulated in QEMU.

7.5. Spacecraft Bus Simulation

The spacecraft bus simulator simulates everything outside of Swivel's flight control unit: the SpaceWire bus, the mission clock, the magnetometer instrument, the telecommunications radio, the background magnetic field, and even mission control. SwivelSim provides a component-based deterministic simulation framework in Golang, which forms the basis for the testbench simulator. [32] The simulator manages the flow of time by allowing components to register timers on a central list, and advances through time in response to messages sent through the timesync protocol by QEMU; it communicates back the timestamp of the next unexpired timestamp to QEMU to ensure that QEMU will transfer control back to the bus simulator when it reaches the expiration time for the timer.

The bus simulator implements the second side of the FakeWire link (Section 5.1.1. FakeWire Protocol) provided by the flight software. It decodes packets immediately on reception, and passes them to the simulated SpaceWire switch, which passes packets to the RMAP handlers for the clock, magnetometer, and radio. The clock only needs to sample the current simulation time to be able to respond to requests. The magnetometer interacts with a deterministic and random simulation of a magnetic field around the spacecraft that bears no resemblance to actual physics, but serves to generate random reading data, which it records for later comparison against the magnetometer telemetry downlinked by the flight software. The radio receives commands from an input generator that generates randomly-selected commands every 200-500 milliseconds; it further decodes and records the telemetry transmitted by the flight software for later comparison.

In order to determine whether the flight software is operating correctly, the testbench simulator tracks and records all material activities that occur in the simulation, including

telemetry downlinked, commands uplinked, and readings taken with the magnetometer. For each requirement listed in 5.2. Flight Software Requirements, the testbench software programmatically tracks all of the material information, and continuously monitors whether the flight software has met the requirements. The verifier generates a log file containing timestamped requirement successes and failures, which allows SwivelSim's analysis tools to later determine whether the flight software was operating correctly within any particular execution period and measure the overall reliability over time.

7.6. Analysis Tools

SwivelSim generates log files that describe the activities that occurred during each experiment. Because the files directly represent the activities that took place during the experiment, they are not readily interpretable by human eyes. Hailburst provides a series of tools to extract, summarize, and render experimental data for human understanding. Section 7.6.1. Viterbi Analyzer describes the tool that extracts statistics from requirement data. Section 7.6.2. Debug Log Extractor briefly describes the debugging tool used to inspect the behavior of Vivid and the flight software. Section 7.6.3. Chart Renderer describes the tool that visualizes the overall behavior of the spacecraft during a simulation.

7.6.1. Viterbi Analyzer

The Viterbi Analyzer tool summarizes the execution of an experiment into a series of statistics. SwivelSim performs this analysis in three stages, starting from the raw requirement success and failure information generated by the simulation, and producing numerical statistics like the ones shown in Table 7-2.

Mean Time to Failure	11.9 seconds
Mean Time to Recovery	0.213 seconds
Best Time to Failure	30.2 seconds
Worst Time to Recovery	0.450 seconds
All-Requirements-Passing %	98.4%

Table 7-2: Sample Reliability Statistics

Individual Requirement Analysis. The first stage translates the raw data points for a single requirement into a summary of the flight software's ability to pass that particular requirement over time. The summary is a sequence of transitions between the states listed in Table 7-3. Figure 7-4 shows an example of this summary on the `ReqReceipt` requirement in a specific trial: at first, the requirement is consistently passing, so SwivelSim assigns it the Passing state. Then, the requirement fails four times in a row, so SwivelSim assigns it the Failing state. Finally, the requirement starts passing again, and SwivelSim assigns it the Passing state again. SwivelSim defines a list of approximate probabilities that the flight software would transition between each of the states listed in Table 7-3, and the approximate probabilities that it would observe different outcomes based on those states, such as a requirement failure, a requirement success, or the absence of a success or failure. It passes these probabilities and the observed requirements to an implementation of Viterbi's algorithm, which computes the most probable sequence of hidden states that would produce the series of observations provided. [33]

Requirement State Name	**Meaning**
Passing	A period of time in which the flight software consistently passes this requirement.
Partial	A period of time in which the flight software intermittently passes this requirement.

Absent	A period of time in which the flight software neither succeeded nor failed at this requirement, because it did not apply.
Failing	A period of time in which the flight software is consistently failing this requirement.
Degrading	A period of time where the flight software is transitioning from Passing to Failing.
Recovering	A period of time where the flight software is transitioning from Failing to Passing.

Table 7-3: Individual Requirement Hidden States

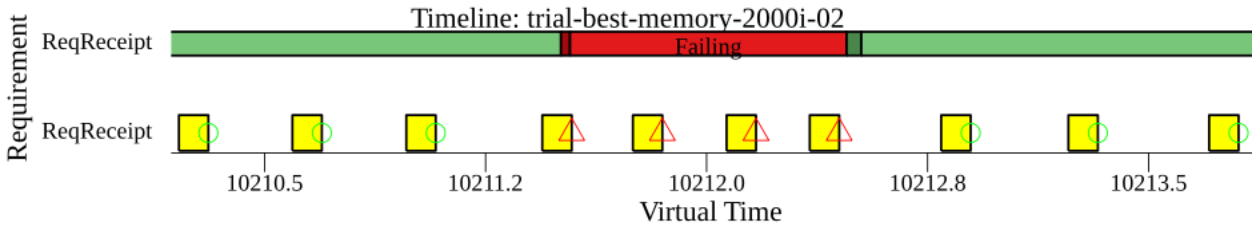


Figure 7-4: Analysis of a Single Requirement

This diagram shows two views of the same requirement successes and failures over a period of around three seconds. The bottom view shows the periods in which the requirements were being evaluated as yellow blocks, followed by a green circle or red triangle indicating whether the requirement succeeded or failed, respectively. The top view displays a summary of requirement success statuses, moving from a green "success" region at the left, through a short intermediate "degrading" region into a red "failing" region in the middle, and then back through a short intermediate "recovering" region before reaching a green "success" region at the right.

System Summary Generation. The second stage of the analysis rolls up the summaries for each individual requirement into an overall summary of the behavior of the flight software as a whole. Figure 7-5 provides a simplified visual example of this process: SwivelSim looks at the periods in which each requirement was determined to most likely have been each hidden state, and fills in the gaps to decide what the period was that the overall system was most likely Broken or Working. In the example shown, SwivelSim concludes that, even though there's a period of a few hundred milliseconds between `ReqCollectMagReadings` failing and `ReqReceipt` failing where there is *technically* no failure reported, it was still most likely the case that the flight software was malfunctioning for the whole time.

This summary is also computed through an application of Viterbi's algorithm, but now the hidden states are just Broken and Working, and the observations are the summary states for each of the requirements. Notably, SwivelSim specifies that there is a 0% chance that the system is Working if any of the requirements have any state besides Absent or Passing; the main purpose of Viterbi's algorithm is to decide what to do about the periods that are ambiguous. (Such as because all the requirements are Absent or Passing at the exact current time, but they aren't before or after the current time.)

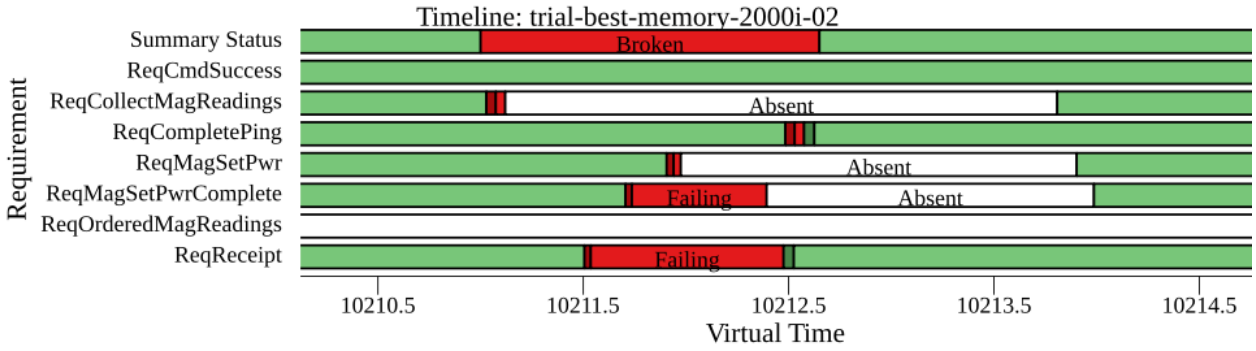


Figure 7-5: System Requirement Summary

This diagram shows a Summary Status of an overall system, and then the series of individual requirement states used to compute that summary. Each row describes a series of transitions between states over around a four-second period. The most prevalent states shown are Broken or Failing (red) and Working or Passing (green).

Statistics Calculation. Once SwivelSim has generated the summary status intervals for the complete simulation duration, it summarizes it into the reliability statistics shown in Table 7-2. Mean Time to Failure represents the average length of all of the intervals where the summary status is the Working state; Mean Time to Recovery represents the average length of all of the intervals where the system is in the Broken state. Best Time to Failure and Worst Time to Recovery are the longest observed intervals where the system is working correctly or not working correctly, respectively. All-Requirements-Passing % is the percentage of the simulation duration in which the system was working correctly. Section 9. Evaluation uses these statistics to compare the reliability of different configurations of Vivid.

```
$ go run ./ctrl/debuglog/ trial-last/guest.log --loglevel debug --from 10015.2 --until 10015.614
[10015.221784715] [     TELEMETRY] [DEBUG] [0] Command Received: Time=10015183501652 CommandId=01000001
[10015.221884715] [     TELEMETRY] [DEBUG] [1] Command Received: Time=10015183501652 CommandId=01000001
[10015.221984715] [     TELEMETRY] [DEBUG] [2] Command Received: Time=10015183501652 CommandId=01000001
[10015.222454747] [     TELEMETRY] [ INFO] [0] Pong: PingId=e9741ac4
[10015.222455947] [     TELEMETRY] [DEBUG] [0] Command Completed: Time=10015183501652 CommandId=01000001
[10015.222464747] [     TELEMETRY] [ INFO] [1] Pong: PingId=e9741ac4
[10015.222465947] [     TELEMETRY] [DEBUG] [1] Command Completed: Time=10015183501652 CommandId=01000001
[10015.222474747] [     TELEMETRY] [ INFO] [2] Pong: PingId=e9741ac4
[10015.222475947] [     TELEMETRY] [DEBUG] [2] Command Completed: Time=10015183501652 CommandId=01000001
[10015.276118731] [     TELEMETRY] [DEBUG] [0] Heartbeat
[10015.276128747] [     TELEMETRY] [DEBUG] [1] Heartbeat
[10015.276138747] [     TELEMETRY] [DEBUG] [2] Heartbeat
[10015.399163755] [     TELEMETRY] [DEBUG] [0] Heartbeat
[10015.399173755] [     TELEMETRY] [DEBUG] [1] Heartbeat
[10015.399183755] [     TELEMETRY] [DEBUG] [2] Heartbeat
[10015.403314587] [      SCRUBBER] [DEBUG] Scrub cycle complete.
[10015.406369595] [      SCRUBBER] [DEBUG] Scrub cycle complete.
[10015.406442619] [      SCRUBBER] [DEBUG] Beginning scrub cycle (baseline kernel ELF at 0x00000c48)...
[10015.409497611] [      SCRUBBER] [DEBUG] Beginning scrub cycle (baseline kernel ELF at 0x00000c48)...
[10015.522208731] [     TELEMETRY] [DEBUG] [0] Heartbeat
[10015.522218747] [     TELEMETRY] [DEBUG] [1] Heartbeat
[10015.522228747] [     TELEMETRY] [DEBUG] [2] Heartbeat
[10015.611545083] [      SCRUBBER] [ WARN] Detected mismatch in read-only memory. Beginning corrections.
[10015.611614795] [      SCRUBBER] [ WARN] Summary for current scrubber step: 2283 word(s) corrected.
[10015.611708891] [         CRASH] [ CRIT] UNDEFINED INSTRUCTION occurred in clip 'sc_watchdog_monitor_clip'
[10015.611711131] [         CRASH] [ CRIT] Status: PC=0x4012f5bc SPSR=0x80000192
[10015.611711899] [         CRASH] [ CRIT] Possible causes: InKernel=1 GlobalRecurse=0 TaskRecurse=0
[10015.611712779] [         CRASH] [ CRIT] Registers:  R0=401500a8  R1=40150098  R2=a27f2b28  R3=4012f5b0
[10015.611713787] [         CRASH] [ CRIT] Registers:  R4=00000001  R5=00000003  R6=a27f5238  R7=00000003
[10015.611714795] [         CRASH] [ CRIT] Registers:  R8=a27f2b28  R9=00000003 R10=40146700 R11=42150098
[10015.611715803] [         CRASH] [ CRIT] Registers: R12=3a27f2ba
[10015.611717083] [         CRASH] [ CRIT] HALTING RTOS IN REACTION TO UNDEFINED INSTRUCTION
```

Figure 7-6: Example extract of debug log generated by Hailburst
(output modified to fit on one page)

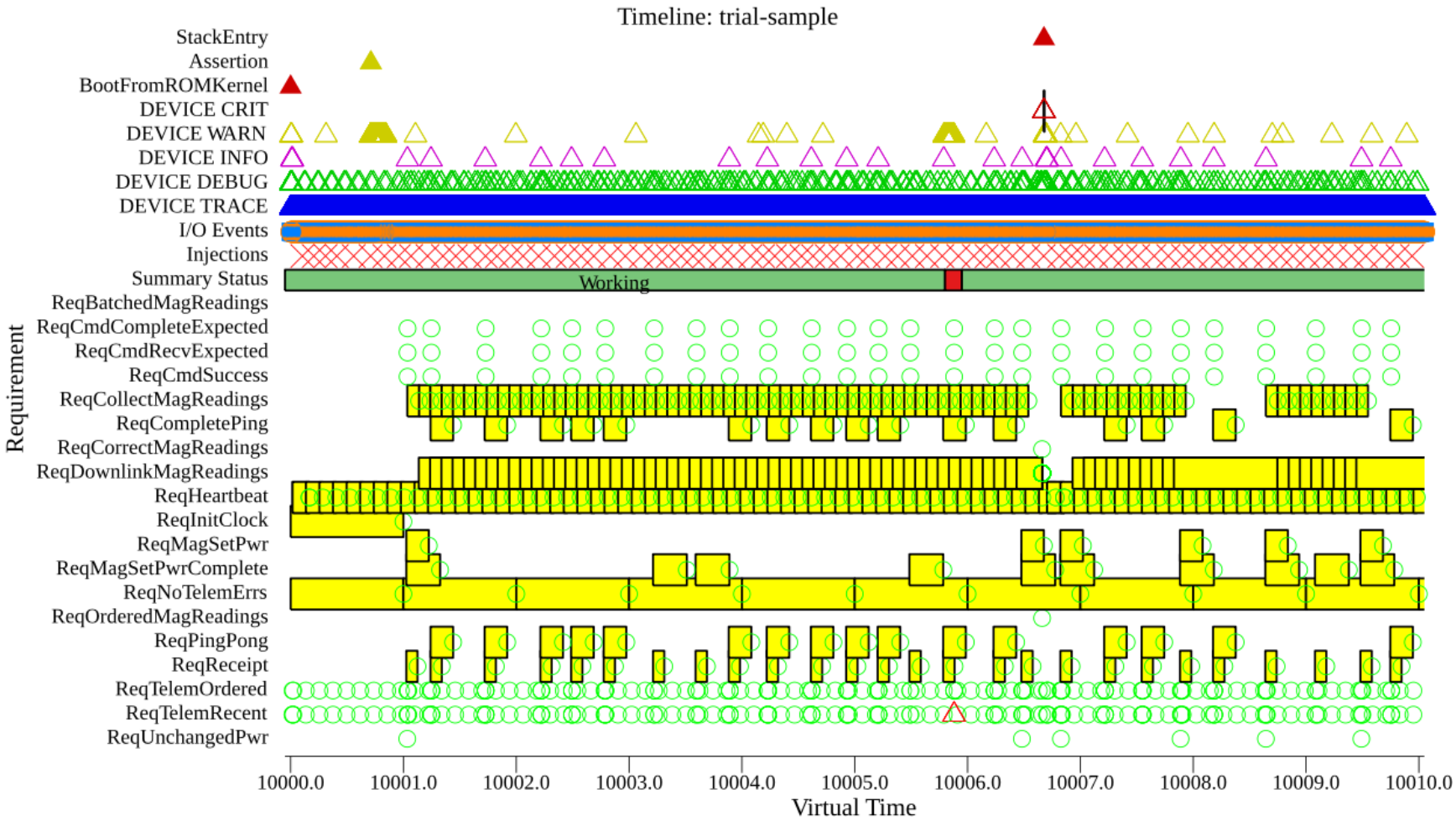


Figure 7-7: Example rendered chart of an experimental trial

This diagram summarizes ten seconds of an experiment trial as a stack of timelines in several sections. The first eight timelines represent logging messages produced by Vivid or the flight software; these are colored by log level: red for critical, yellow for warning, purple for info, green for debug, and blue for trace. The next timeline shows I/O Events: orange for messages sent by the flight software, and blue for messages sent to the flight software. The Injection timeline shows a red X every time Hailburst injected a fault. The System Summary timeline the times when the Viterbi Analyzer decided that flight software was Working (green) or Broken (red). The remaining timelines describe the raw requirement successes and failures observed for each spacecraft behavior requirement; green circles reflect successes, red triangles reflect failures, and yellow blocks reflect intervals where SwivelSim was still deciding whether a requirement was satisfied.

7.6.2. Debug Log Extractor

The debug log extractor analyzes the binary-packed debug logs generated by Vivid software and produces a full-text rendering, like the one shown in Figure 7-6. The tool allows an engineer to filter by multiple features of the logs, including log severity level and time period, which helps engineers inspect specific periods of time more closely. Section 4.1. Debug Log System describes how the debug log parser extracts the information displayed by this tool.

7.6.3. Chart Renderer

The chart renderer combines and summarizes all of the different data sources available from an experiment and displays them in a single stack of timelines, as demonstrated in Figure 7-7. The render displays the requirement success/failure observations recorded by Hailburst during an experiment on a per-requirement basis, along with the "Summary Status" line generated by the Viterbi Analyzer. Hailburst also displays the times at which different injections occurred, all of the input and output events that occurred on the timesync connection, and the different severities of debug logs produced by the flight software. The caption on Figure 7-7 explains how the chart diagram renders each of these details.

8. The Hailburst Fault Injection System

Hailburst is a tool for continuously and precisely injecting a series of radiation faults into a running QEMU emulator without a significant loss of emulation performance, such as the emulator used by SwivelSim (Section 7. The SwivelSim Avionics Simulation). Section 8.1. Fault Model discusses the fault model for the radiation faults that Hailburst injects and Section 8.2. Precise Fault Injection describes how the fault injection process works.

8.1. Fault Model

Radiation can translate into a wide range of different physical faults in processor circuits. Some of these faults result in permanent radiation damage, some result in short-term disruptions or burnouts that can only be resolved by a power cycle, and some cause only a transient upset to a single state within the circuit. [34] Out of these, Vivid only intends to address individual transient upsets, not any of the rarer and more complex forms of faults. Although processors can internally experience faults in a number of forms, they will generally resolve into a discrepancy in one or more bits of state in the processor or memory, which will typically be equivalent to the impact of a single bitflip. Therefore, Hailburst seeks to inject only single bitflips into registers and memory, simulating the most common forms of radiation faults.

Because the relative ratios of different types of injections that will occur in practice are difficult to predict, Hailburst distributes memory and register injections uniformly over the available memory and register locations. Hailburst is generally configured either for memory injections or register injections in a specific experiment, but not both at the same time.

8.2. Precise Fault Injection

Hailburst injects faults into QEMU by attaching a GDB process to it. GDB (the GNU debugger) is a standard piece of GCC toolchain software normally used for interactive inspection and modification of the state of a piece of malfunctioning software. Instead of using GDB to analyze or resolve bugs, Hailburst uses it to create them. GDB can attach to a QEMU's emulated processor through a special stub in QEMU, which implements GDB's remote serial protocol, and Hailburst enables QEMU's PhyMemMode option to ensure that it ignores any memory isolation or virtual addressing features enabled by the operating system.[31]

Command Signature	**Command Behavior**
`listram`	Lists all of the general-purpose memory ranges defined by QEMU, which memory injections might target.
`listreg`	Lists all of the registers exposed by QEMU to GDB, which register injections might target.
`stepvt <time>`	Asks QEMU to advance the simulation clock by the specified number of nanoseconds.
`log_inject <log.csv> [log.gdb]`	Starts logging all injections into a CSV file for later analysis, and optionally a GDB script to replay this sequence of injections.
`inject [addr]`	Injects a bitflip into the specified address, or a random address in QEMU's general-purpose memory ranges if not specified.
`inject_reg [reg]`	Injects a bitflip into the specified register, or a random register exposed by QEMU if not specified.
`task_restart`	Replaces the instruction at the current program counter with the ARM `UDF` instruction, to force the current execution clip to crash.
`task_disrupt [addr]`	Replaces the instruction at the specified address with an ARM `UDF` instruction, to force the next execution clip that runs it to crash. If not specified, picks a random address in the flight software text segment.

[31] Because Vivid does not yet implement memory isolation — see 3.6.3. Missing Safety Features — this is mostly relevant to injecting faults into Linux.

`continuous <iterations> <min> <max> <mode>`	Every `min` to `max` nanoseconds (uniformly distributed), for the specified number of iterations, injects a random fault according to the mode provided: `mem` for the `inject` command, `reg` for the `inject_reg` command, `restart` for the `task_restart` command, or `restart-later` for the `task_disrupt` command.

Table 8-1: The GDB commands provided by Hailburst

Hailburst is structured as a series of GDB commands implemented with GDB's Python API, which are listed in Table 8-1. Most of these commands are implemented in terms of GDB's existing interfaces. For example, the `inject` command uses GDB's interfaces to read a memory address, modify the data by flipping a single bit, and write back the result to the same memory address. However, some of Hailburst's commands require features not exposed by QEMU.

The `stepvt` command, in particular, is challenging to implement. Normally, the remote serial protocol supports only single-stepping, watchpoints, and breakpoints; there is no efficient mechanism to pause and resume QEMU's execution at precise times for fault injection.[32] GDB *does* have a mechanism to interrupt QEMU at its current execution point, but these interruptions can be timed based only on wall clock time, which runs wildly faster and slower (at alternating times) than the simulated execution time in QEMU.[33] Further, QEMU will only check for external interruptions at regular points during execution, so using GDB's normal interruption mechanism would make those execution points "hotspots" for fault injections. To work around these limitations, Hailburst patches to QEMU to add a new monitor command. This command registers a timer at a specific simulated clock time and halts the simulation at that time, which returns control back to GDB. Hailburst's `stepvt` command can invoke this command through

[32] Repeatedly single-stepping is significantly slower than QEMU's full execution speed.
[33] SwivelSim configures QEMU in icount (instruction count) mode, as described in 7.2. QEMU Machine Configuration, which is why simulated execution time can diverge from wall clock time. However, interruption precision would not matter in the first place if SwivelSim disabled icount mode, because the simulation would be completely nondeterministic.

the remote serial protocol before it resumes QEMU's execution; this ensures that QEMU executes only for the precise number of simulated nanoseconds requested before returning control to Hailburst.

The `continuous` command implemented by Hailburst allows engineers to simulate a series of injections into a running system, interleaved at randomly-selected intervals. SwivelSim is configured with a selection of default `continuous` command lines it can execute in Hailburst, to facilitate executing fault injection experiments.

In order to support reproducible injection experiments, where Hailburst performs the same injections at the same times, Hailburst's `log_inject` offers the ability to record the actual injections performed (such as those randomly generated by the `continuous` command) into a GDB script. This script can be replayed precisely in the future using GDB's `source` command, which allows Hailburst to operate fully deterministically when desired; if an engineer runs the same simulation twice, with the same series of injection commands, she can reproduce the exact same series of outcomes.

9. Evaluation

This section seeks to answer four questions:

- Does SwivelFSW function correctly in the absence of radiation? (9.1. Application Correctness)
- Does the Hailburst fault injection system work? (9.3. Fault Injection Demonstration)
- How well does Vivid defend against injected radiation faults? (9.4. Reliability of Vivid)
- How much does Triple Modular Replication contribute to Vivid's resilience to radiation? (9.5. Triple Modular Replication)

This chapter answers these questions by running a sequence of experiments with Hailburst, Vivid or Linux, and the testbench flight software. Section 9.2. Fault Injection Rates explains how the fault injection rates in these experiments were determined. Different experiments enable or disable different sets of Vivid's defenses, as described in Section 4.9. Reliability Configuration.

9.1. Application Correctness

To determine whether SwivelFSW operates normally in the absence of radiation, I simulated it in three different configurations. For the first trial, I compiled it with Vivid with all defenses enabled, and instructed SwivelSim to simulate the system for 15 minutes (of simulation time) without injecting any radiation faults and evaluate the behavior against Swivel's requirements. For the second trial, I performed the same experiment, except that I disabled all of Vivid's defenses. For the third trial, I performed the same experiment again, but I compiled SwivelFSW to run on Linux instead of Vivid.

Figure 9-1 displays a zoomed-out execution chart for the first trial, where Vivid is fully defended. During this trial, all of the requirements listed in Section 5.2. Flight Software Requirements passed every time. SwivelSim evaluated some requirements more frequently, such as `ReqHeartbeat`, which it confirmed passed several times a second because of the frequency of the heartbeat messages; SwivelSim evaluated other requirements less frequently, like `ReqBatchedMagReadings`, because they only mattered when the flight software downlinked magnetometer data, which occurred less frequently. The fact that none of the requirements ever failed during this trial indicates that SwivelFSW running on Vivid in its fully defended configuration satisfies all of the spacecraft requirements — or at least tends to satisfy them for the first fifteen minutes. It is possible that there are rare cases where SwivelFSW will fail some of Swivel's requirements in the absence of radiation, but the first trial indicates that, if this happens, it must be rare.

The results of the second and third trial are shown in Figure 9-2 and Figure 9-3. Hailburst detected zero requirement failures in both cases, the same as the first test. As a result, it is likely that any requirement failures encountered in future tests in any of these configurations can be attributed to the injected radiation faults, rather than due to underlying bugs in SwivelFSW or SwivelSim.

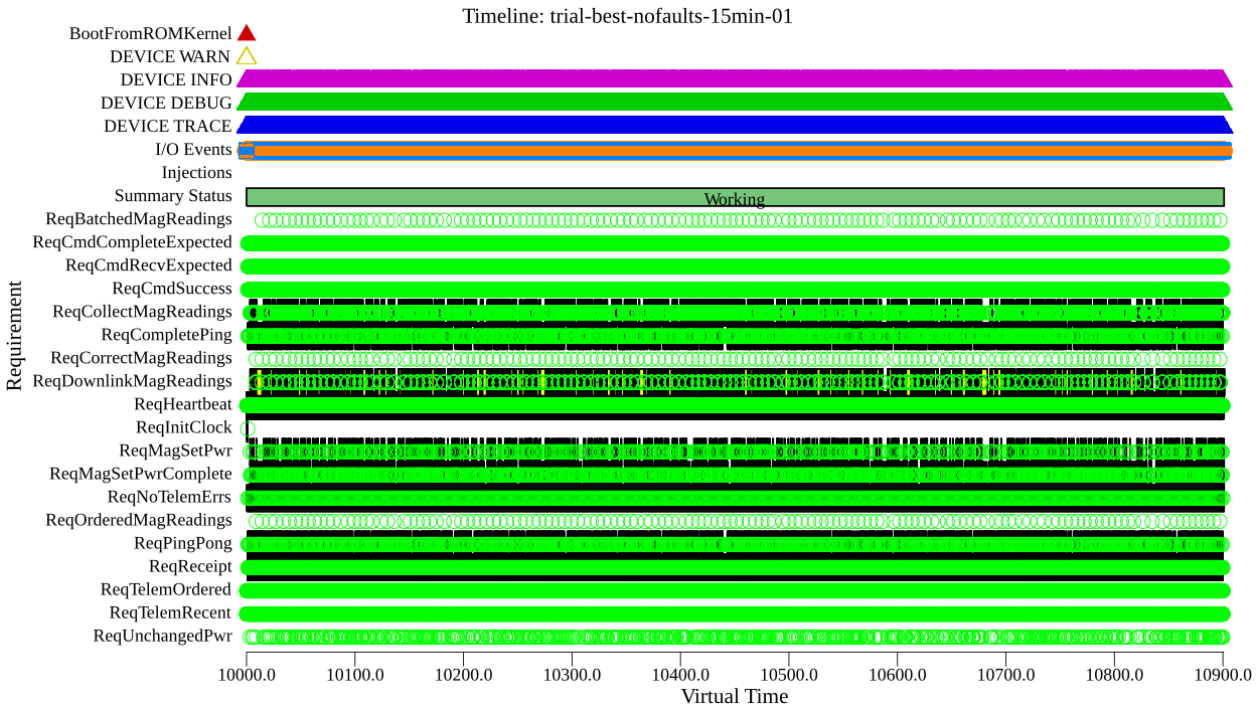


Figure 9-1: Execution of a fully-defended Vivid system for 15 minutes

(See Section 7.6.3. Chart Renderer for an explanation of how to read these charts.)

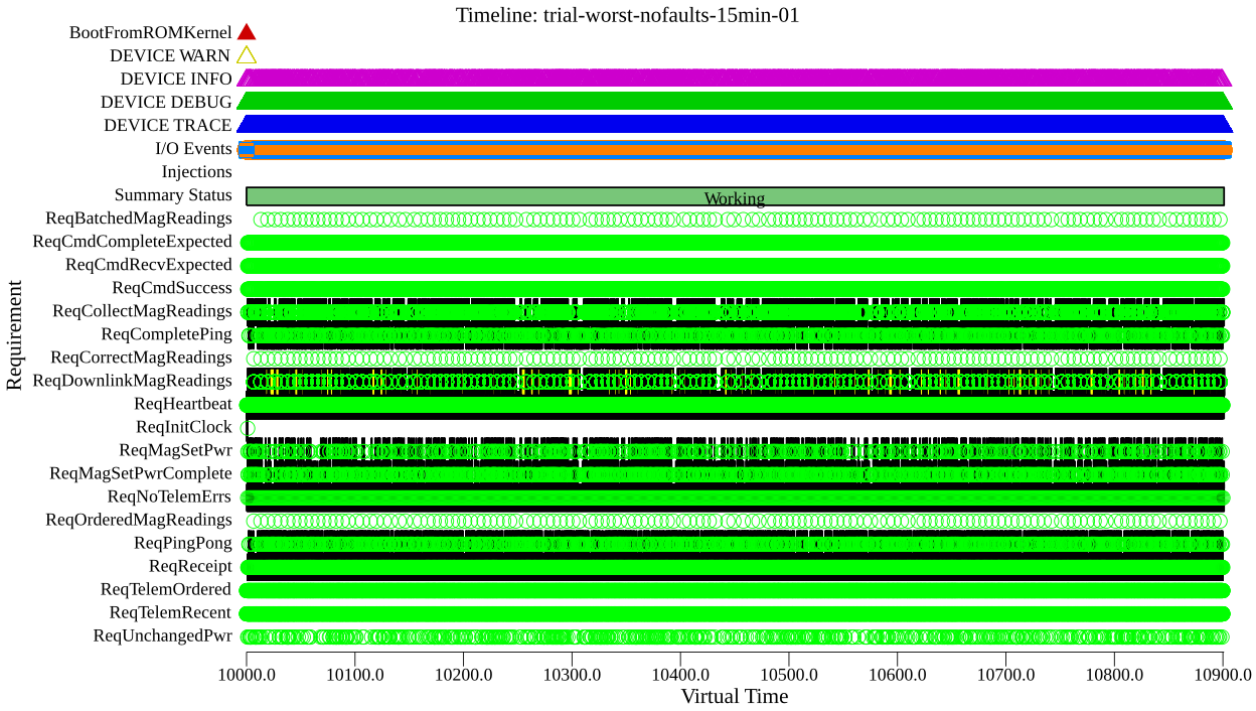


Figure 9-2: Execution of an undefended Vivid system for 15 minutes

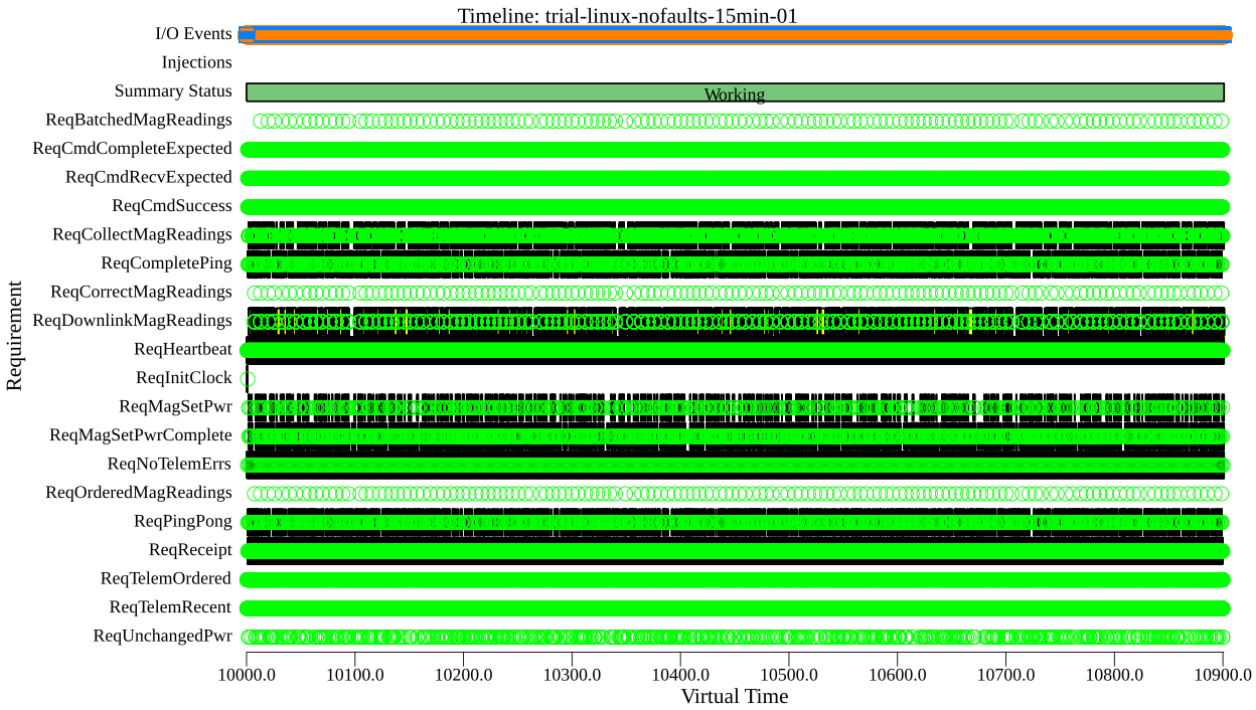


Figure 9-3: Execution of an undefended Linux system for 15 minutes

9.2. Fault Injection Rates

As a real-world example of the radiation levels experienced by an actual spacecraft, the CALIPSO spacecraft's non-radiation-hardened SDRAM experienced an average rate of 0.000058 bitflips per minute per MiB of RAM[34] and 0.000018 processor miscompares per minute per processor[35]. [6] The memory injection rate can be entered as an injection rate for Hailburst, but this would not be useful.[36] Even with SwivelSim and Hailburst's optimizations, they cannot simulate the Swivel spacecraft and the SwivelFSW flight software much faster than real time, even in the best case. (The simulation performance varies from approximately 0.1

[34] CALIPSO reported 493 SEUs across 68 MiB of RAM over 87 days. Dividing 493 by the product of 68 times 87 times 1440 (the number of minutes in a day) yields 0.000058.
[35] CALIPSO reported 9 processor faults across 4 processors over 87 days. Dividing 9 by the product of 4 times 87 times 1440 (the number of minutes in a day) yields 0.000018.
[36] The processor rate cannot be directly translated, because it appears likely that each processor miscompare represents multiple underlying register faults; a miscompare on CALIPSO will only occur if a register fault leads to a discrepancy in the processors' address, data, or control lines, which will not necessarily occur for all register faults.

simulated seconds per real second to approximately 2.5 simulated seconds per real second.) If this chapter used realistic injection rates, injecting a thousand faults (a smaller number) would take around ten years. Therefore, it is necessary to inject radiation faults at an accelerated rate.

To select radiation injection rates, the goal is to balance two factors: first, more frequent fault injections will translate into faster experiments, which allow for the collection of more significant quantities of data in the same amount of time; but second, more frequent fault injections will increase the chance that two faults arrive at unrealistically close times and cause vote splits or other failures that would not have occurred if Vivid had more time to recover between fault injections.

In this evaluation, injections are always either distributed over the range of 100 milliseconds to 150 milliseconds, or 200 milliseconds to 300 milliseconds. Because Vivid's scrubber normally takes approximately 500 milliseconds per scrubbing cycle, and other recovery mechanisms should take comparably long to repair errors in mutable memory, an interval of at least 100 milliseconds ensures that no more than a few faults are likely to be outstanding at the same time. Because there is a higher probability of a register corruption damaging a critical piece of state, compared to a memory corruption, some experiments use a lower rate of register injection than memory injection.

9.3. Fault Injection Demonstration

To determine whether Hailburst can simulate radiation faults in registers and memory that might cause operating systems to malfunction, I ran six experiments where I injected faults into a Linux system running the testbench flight software. Three trials injected faults into registers, and

three trials injected faults into memory. The trials injected up to 1000 faults each, spaced apart by uniformly distributed simulation times in the range 200 milliseconds to 300 milliseconds.

Figure 9-4 summarizes the execution for one of the register injection trials; Figure 6-5 summarizes one of the memory injection trials. In both examples, the Summary Status timeline transitions from Working (green) to Broken (red), indicating that the Linux system operated correctly for a number of seconds, and then failed permanently and did not subsequently recover. This makes sense, because Linux does not have a driver for Vivid's custom watchdog, and the watchdog therefore had to be disabled; without a watchdog, once the operating system irrevocably malfunctions, the system cannot repair itself. In Figure 9-4, the system operated correctly for less than two seconds. In Figure 9-5, the system operated correctly for around 81 seconds.

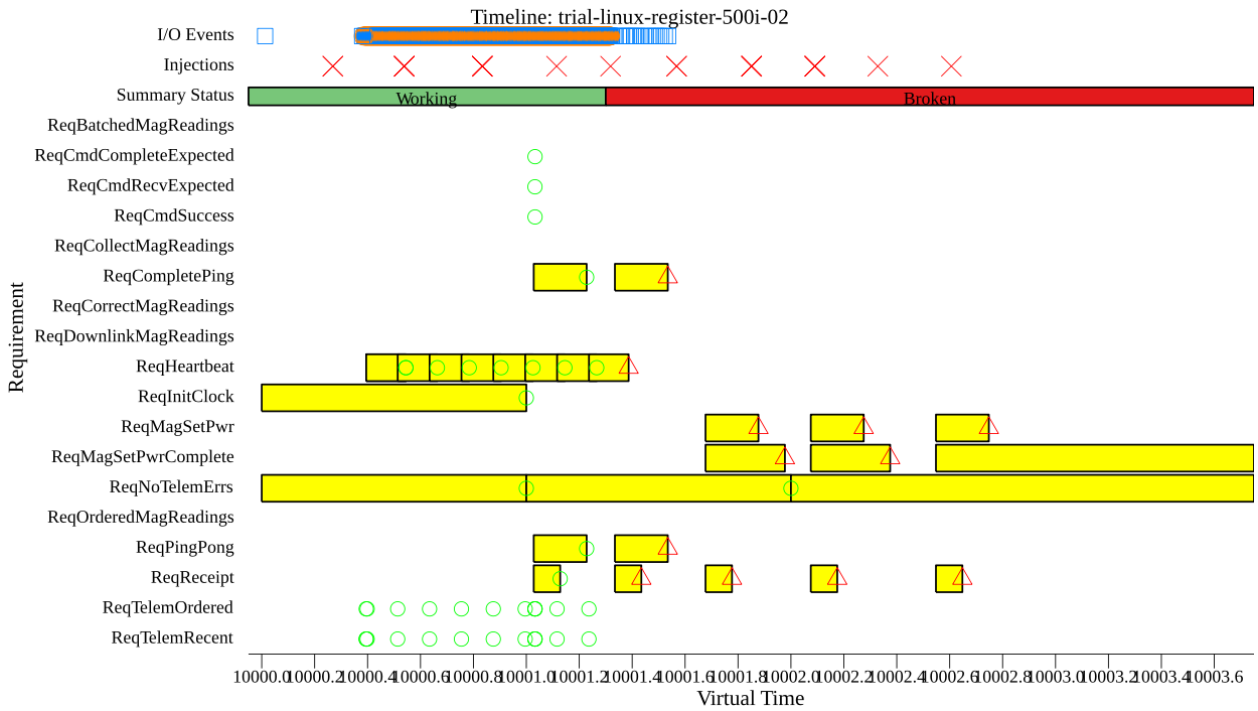


Figure 9-4: A register injection trial for Linux

(See Section 7.6.3. Chart Renderer for an explanation of how to read these charts.)

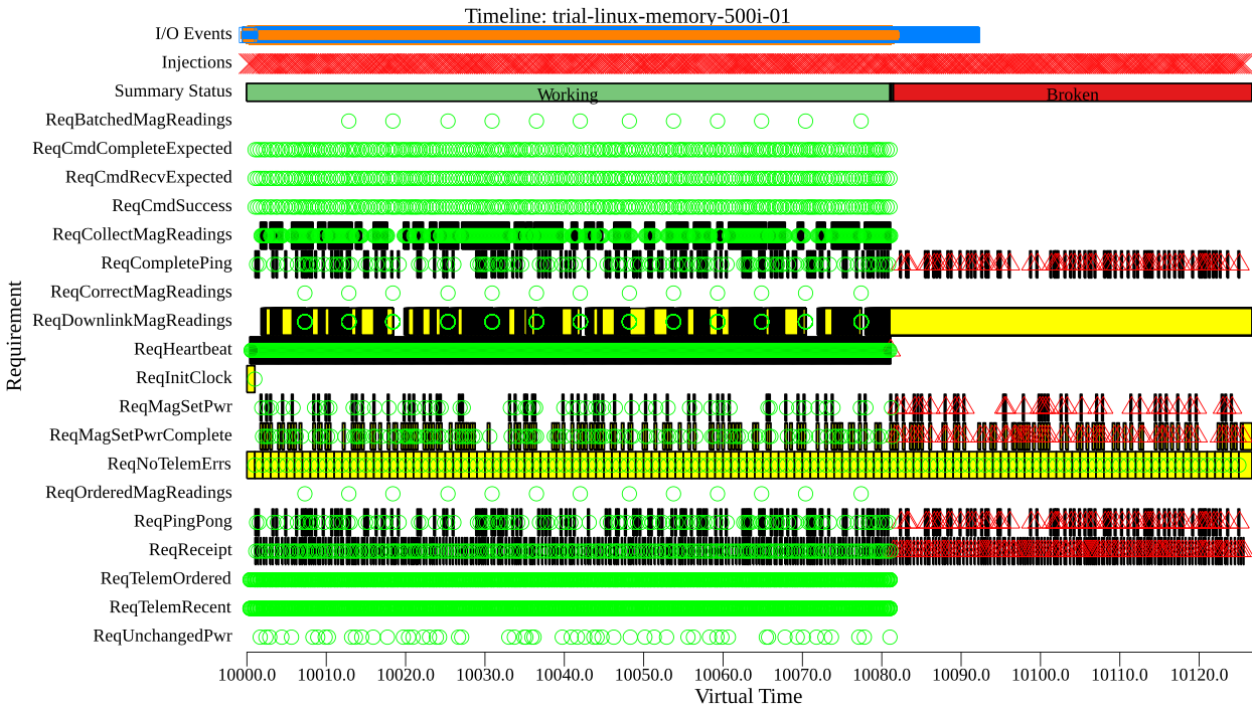


Figure 9-5: A memory injection trial for Linux

Table 9-6 lists the times and qualitative descriptions of the failures detected during each of these trials. In every memory injection case, the Linux system failed permanently within 205 seconds and did not recover. In every register injection case, it failed permanently within 4 seconds. The register faults cause Linux to crash more quickly, because a significant portion of them will corrupt the current executing state of the kernel's idle task (the task that runs most frequently), which is not fault-tolerant; Linux cannot keep operating after Hailburst injects a register error into the idle task, so the system quickly crashes. The memory faults take longer because the probability of any individual bitflip affecting a critical portion of the kernel's 20 MiB of memory is small, so it takes longer for any fatal errors to occur. However, the memory trials always eventually crash, because Hailburst will eventually inject a memory fault into a region of memory that matters.

Section 9.1. Application Correctness demonstrated that Linux operates correctly for at least 15 minutes in the absence of radiation errors, and the specific errors listed in Table 9-6 are consistent with corruption of the kernel's data structures and current execution state; therefore, I conclude that Hailburst can successfully mimic the effects of radiation faults.

Trial	Failure Time	Type of Failure
Memory-1	81 seconds	Null pointer dereference in kernel
Memory-2	75 seconds	Silent crash for unknown reason
Memory-3	205 seconds	Page domain fault in IRQ handler
Register-1	4.0 seconds	Silent crash after stack pointer corruption
Register-2	1.3 seconds	Silent crash after stack pointer corruption
Register-3	1.5 seconds	Unable to handle kernel paging request

Table 9-6: Summary of Linux failures

9.4. Reliability of Vivid

To evaluate how reliable Vivid is, as a result of its defenses, this section compares two configurations of Vivid: one configuration has every defense in Section 4.9. Reliability Configuration turned on (All Defenses), and the other has every defense turned off, except for `Options.NoWatchdog` (Watchdog Only).[37] This section presents Mean-Time-to-Failure, Mean-Time-to-Recover, and All-Requirements-Passing-% statistics for twelve experimental trials, which include three memory injection trials and three register injection trials for each configuration. (The statistics are computed as described in Section 7.6.1. Viterbi Analyzer.)

[37] Without the watchdog enabled, Vivid, like other operating systems, will eventually crash in a way that prevents recovery, which would limit the usefulness of the averaged statistics.

For the memory trials, Hailburst injects 2000 memory faults into random addresses in the 3 MiB of main memory at intervals uniformly distributed between 100 and 150 milliseconds. For the register trials, Hailburst injects 2000 register faults into random registers at intervals uniformly distributed between 200 and 300 milliseconds. (Hailburst selects randomly from the list of registers exported by QEMU that Hailburst can inject through the GDB interface.)

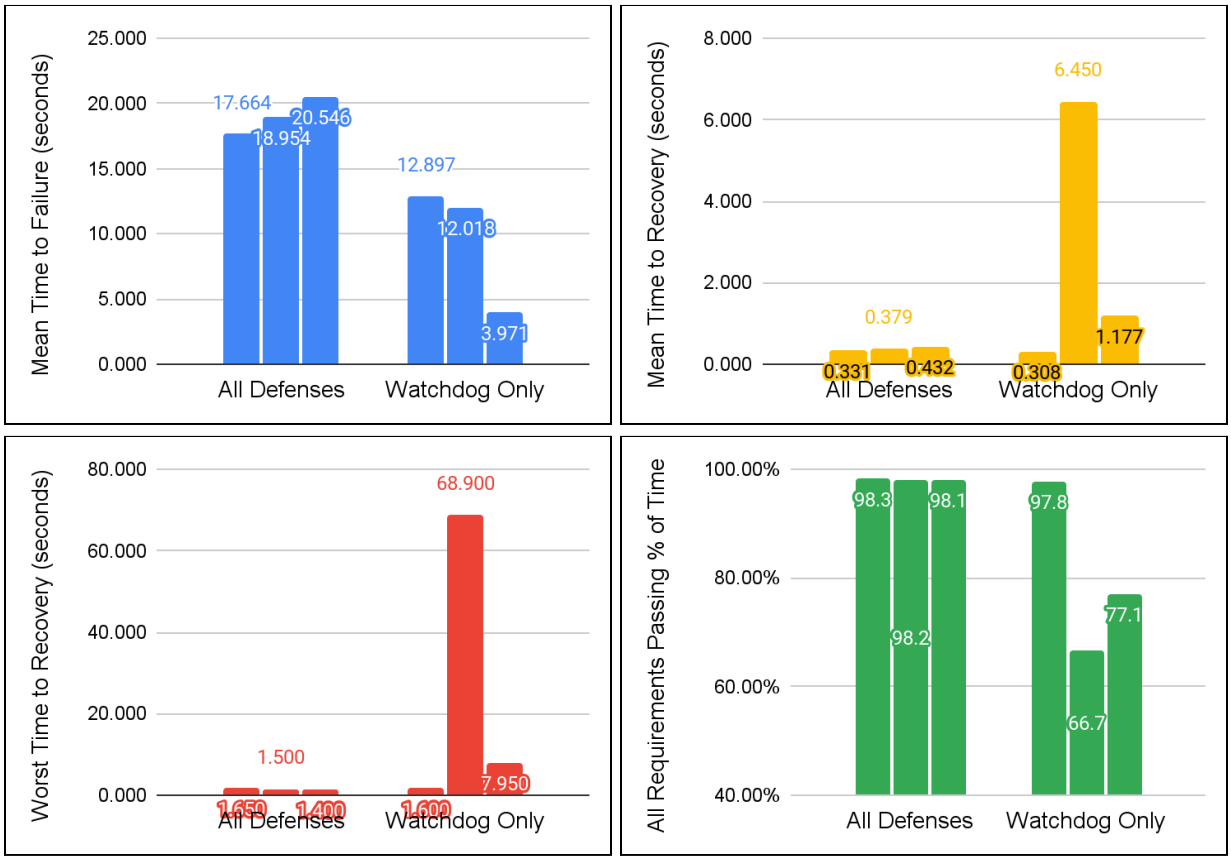


Figure 9-7: Memory Trial Statistics

In order, these diagrams show the Mean Time to Failure, Mean Time to Recovery, Worst Time to Recovery, and All Requirements Passing % of Time, as shown on the vertical axis, for six trials. Each of the six bars represents one trial, clustered into the three trials for the All Defenses configuration and the three trials for the Watchdog Only configuration.

Figure 9-7 compares the memory injection statistics of the All Defenses and Watchdog Only configurations. Because 2000 injections is small compared to the amount of memory, there are discrepancies within the results for each configuration, but the overall trend is clear: when all

defenses are enabled, Vivid lasts several seconds longer on average between visible failures caused by radiation, compared to when all defenses are disabled. (The statistics do not distinguish between complete crashes and partial failures.) When a failure does occur, Vivid's defenses are effective against memory errors: it fully recovers on average within a few hundred milliseconds; without the defenses, Vivid may take much longer to recover; in the worst trial, Vivid averaged over six seconds to recover from a radiation error, and spent nearly 40% of its execution time failing one or more requirements.

The results of the memory trial indicate that Vivid partially succeeds at its goal of seamlessly defending against nearly every possible radiation fault; it experiences a visible failure about once for every 150 injected memory faults, which is an improvement over undefended Vivid, but not by the margin desired. Undefended Vivid experiences a visible failure about once every 30 to 100 injected memory faults.

Vivid succeeds, however, at its goal of reliably recovering from radiation faults: the average-case recovery delay for fully-defended Vivid was around 400 milliseconds, and even the worst recovery delay observed for fully-defended Vivid was 1.65 seconds. (Because Vivid's watchdog can be fed up to 1 second early, and expires 1 second after the end of the feeding period, worst-case recovery delays of up to 2 seconds are inescapable; see Section 10. Future Work and Summary for discussion on reducing this further.) By contrast, the worst-case observed recovery delay for undefended Vivid was 68.9 seconds, and even recovery after 68.9 seconds may only have occurred as a result of an unrelated crash forcing the system to reinitialize.

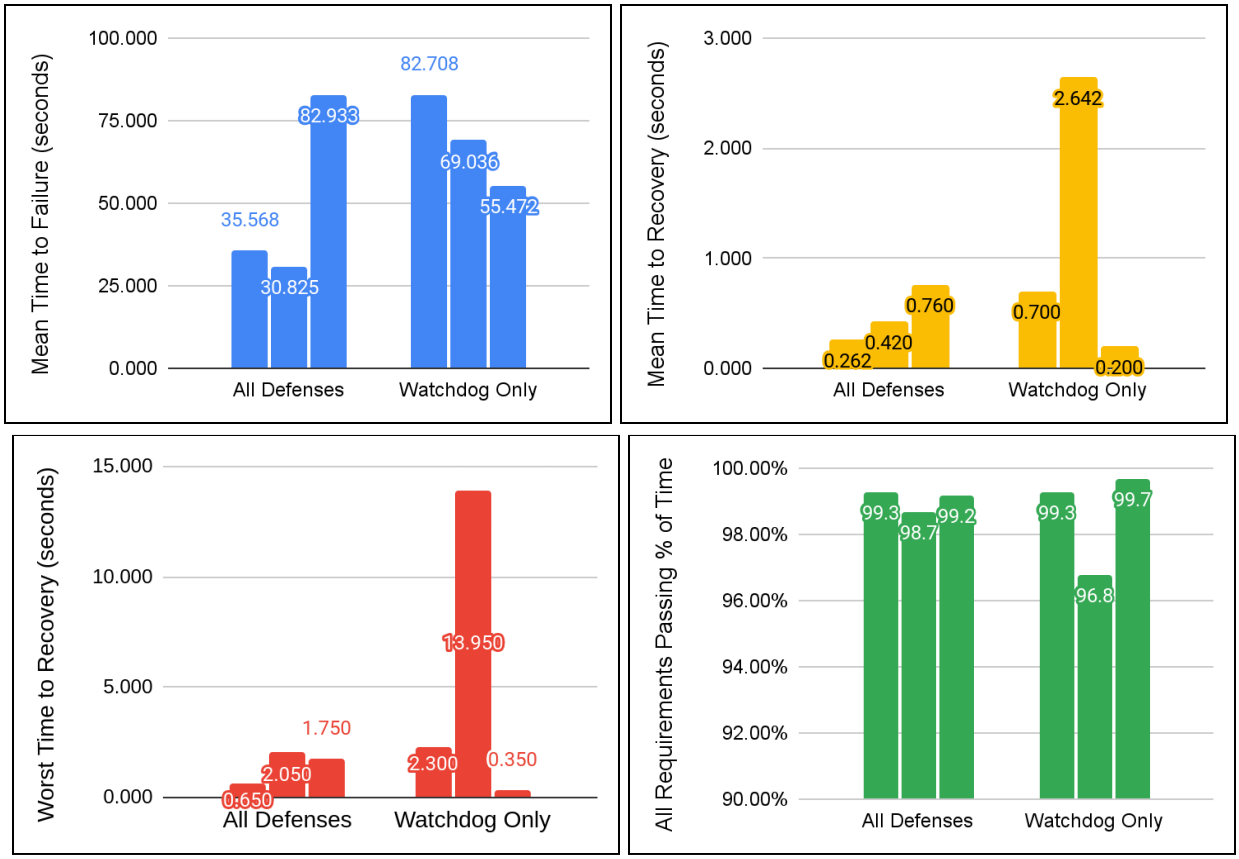


Figure 9-8: Register Trial Statistics

Figure 9-8 compares the register injection performance statistics of the All Defenses and Watchdog Only configurations. These graphs show slightly different results from the memory injection statistics: notably, the Watchdog Only configuration lasts longer, on average, between visible failures that result from radiation. This makes sense, because with fewer replicas of each component running, the processor will spend less time performing meaningful work; any register injections that occur while the processor is waiting on the next timer interrupt will be meaningless, because the partition scheduler throws away all register and stack state when it switches to a new clip. This indicates that Vivid's triple-replication design may overall *increase*

the vulnerability of flight software to register faults, because it will reduce the processor idle time, which is the workload most easily defended against radiation faults.[38]

Like for the register trials, Vivid's defenses do improve the flight software's recovery times. The worst case observed recovery time for fully-defended Vivid was 2.05 seconds (in line with the time expected for the watchdog to force a reset), whereas the worst case observed recovery time for undefended Vivid was 13.950 seconds. The average recovery time for fully-defended Vivid was around 300-800 milliseconds, rather than the average recovery time of up to 2700 milliseconds for undefended Vivid.

Combining both sets of results, it appears that the fully defended configuration of Vivid passes all requirements more than 98% of the time, even in the presence of a radiation level that, as described in Section 9.2. Fault Injection Rates, exceeds realistic radiation levels by several orders of magnitude.

The statistics in this section do not distinguish between partial and complete system failures, because the requirement-tracking system used by Hailburst does not readily distinguish between the two cases. They also do not distinguish between silent data corruptions and crashes.

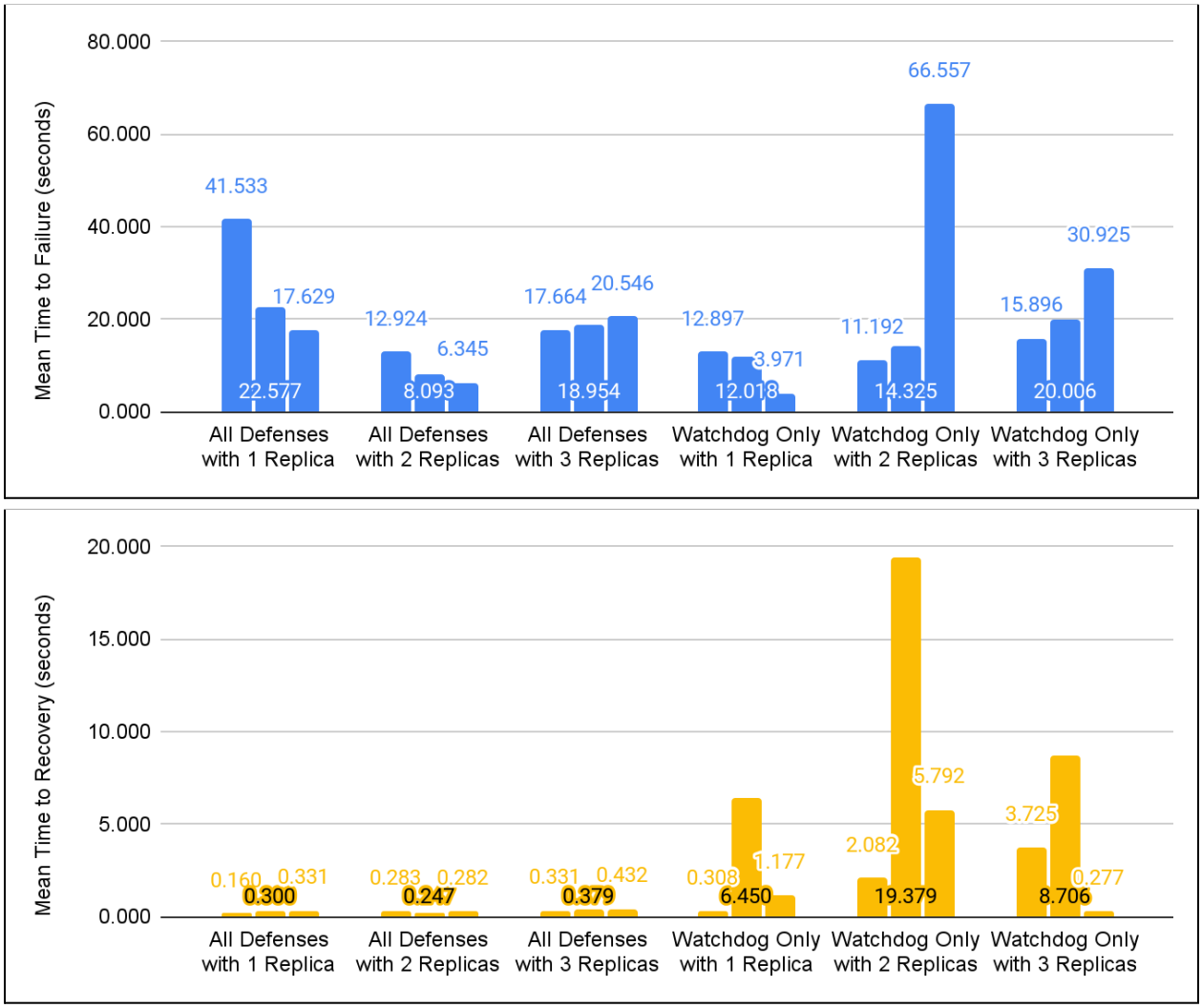
9.5. Triple Modular Replication

Vivid's design, as given in Section 3. Design of the Vivid Kernel, focuses on support for triple-modular replication, but the results of the register injection trials in Section 9.4. Reliability of Vivid suggested that the increased CPU load might have downsides in terms of the overall

[38] While it is easy to defend idle work against radiation faults, Section 9.3. Fault Injection Demonstration indicated that Linux is vulnerable to errors in the idle task. Because the Watchdog Only configuration still benefits from some of Vivid's design features that cannot be disabled through configuration options, the comparison between Watchdog Only and All Defenses may not accurately reflect the advantages and disadvantages of Vivid as a whole.

reliability of the system. This section seeks to determine what the impact of replication is on Vivid, independent of the other defenses.

This section expands the memory and register injection testing described in Section 9.4. Reliability of Vivid by adding four additional configurations to make a total of six. The new configurations include two variants on the All Defenses configuration with one replica (no replication) and two replicas (comparison, but no correction) and two variants on the Watchdog Only configuration with two replicas and three replicas. Figure 9-9 displays the memory trial results for all six configurations, and Figure 9-10 displays the register trial results.



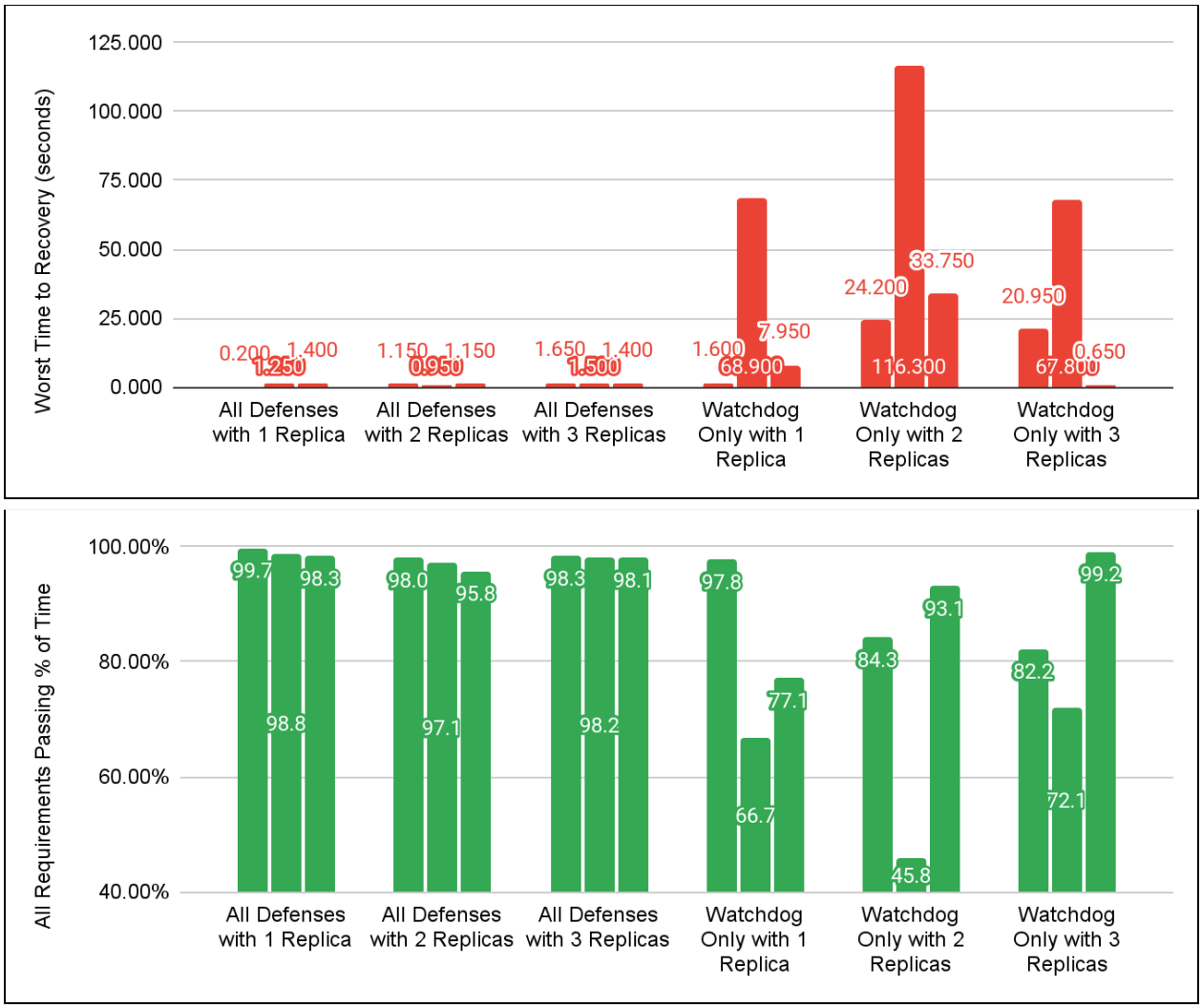


Figure 9-9: Memory Fault Injection Results

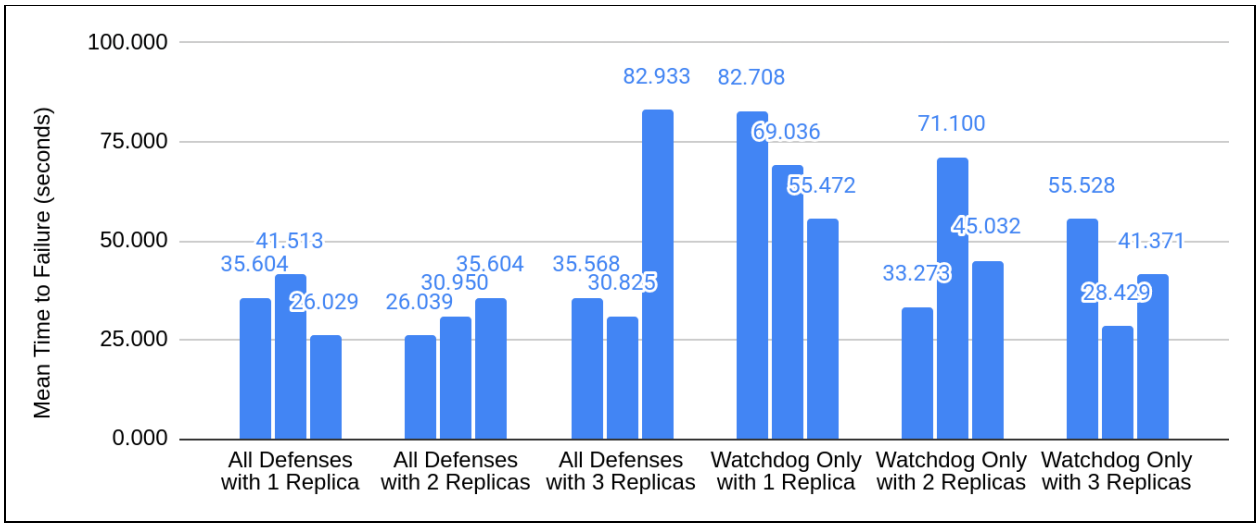
In order, these diagrams show the Mean Time to Failure, Mean Time to Recovery, Worst Time to Recovery, and All Requirements Passing % of Time, as shown on the vertical axis, for eighteen trials. Each of the eighteen bars represents one trial, clustered into three trials each for each of the six configurations labeled on the horizontal axis.

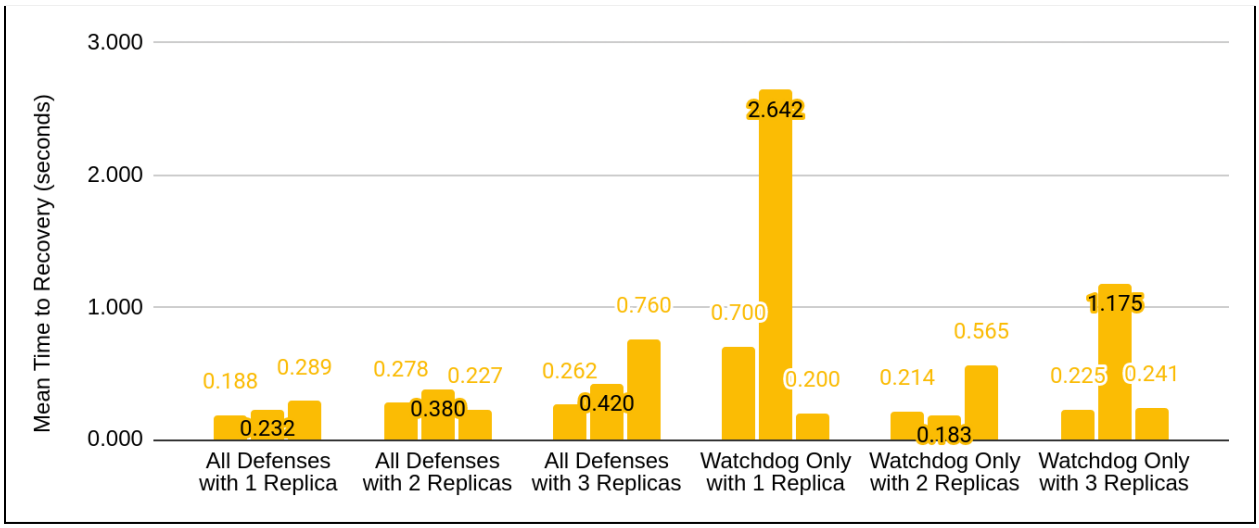
Based on the memory trial results in Figure 9-9, the worst number of replicas, in terms of Mean Time to Failure and All Requirements Passing %, is the 2-replica configuration, regardless of the state of other defenses. With one exception, they fail sooner on average, and pass their requirements a smaller % of the time. This makes sense, because they have a higher probability of experiencing a radiation fault than a 1-replica configuration, but they do not have the higher

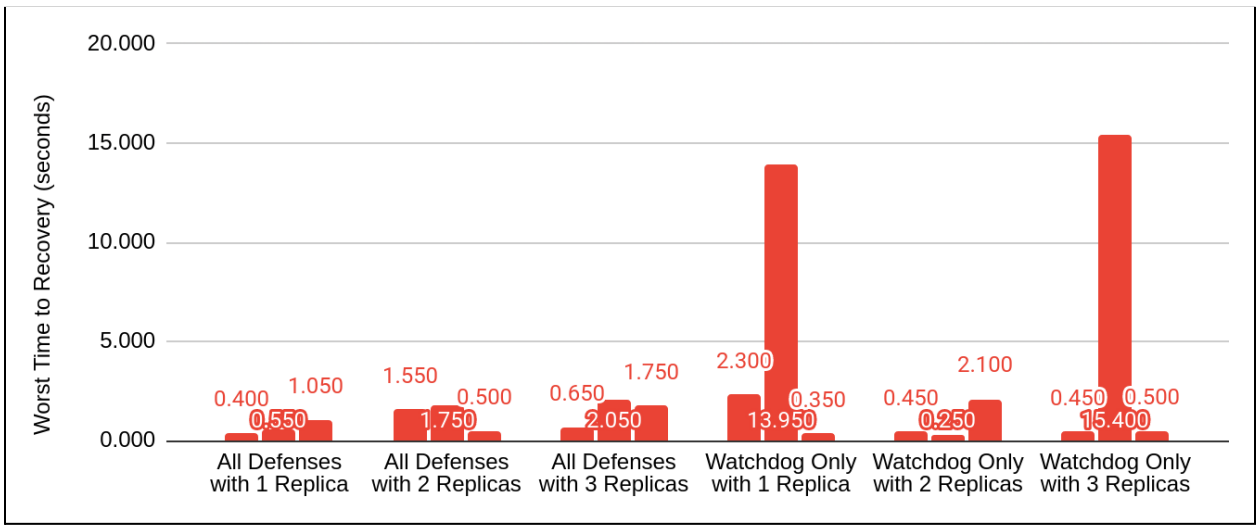
chance of survival that a 3-replica configuration has; if one of the two replicas fails, Vivid will detect that they don't match, and drop all messages they try to send, because there is no majority. In theory, the 2-replica configuration should have fewer silent data corruptions than the 1-replica configuration, but this does not outweigh the increased vulnerability.

In the All Defenses memory trial results, the 1-replica configuration counterintuitively performs slightly (though not significantly) better than the 3-replica configuration: it has a higher Mean Time to Failure, a lower Worst Time to Recovery, and a higher All Requirements Passing %. This likely indicates that the reliability gains from triple-replication are offset by higher probabilities of memory injections hitting in-use memory. However, this result does not measure the nature of the failures; it is possible that the 3-replica configuration trades off higher crash rates for lower silent data corruption rates.

In the Watchdog Only results, the 3-replica configuration performs better than the 1-replica configuration, with a higher Mean Time to Failure, a higher All Requirements Passing %, and comparable Worst Time to Recovery. This may indicate that one of the other defenses (such as code replication) may be responsible for sabotaging the 3-replica results in the All Defenses case. Section 10. Future Work and Summary will discuss future steps for further analyzing this result, and some possible approaches for improving the reliability of 3-replica configurations.


Mean Time to Failure (seconds)
100.000
75.000
50.000
25.000
0.000
35.604
41.513
26.029
26.039
30.950
35.604
35.568
30.825
82.933
82.708
69.036
55.472
33.273
71.100
45.032
55.528
28.429
41.371
All Defenses with 1 Replica
All Defenses with 2 Replicas
All Defenses with 3 Replicas
Watchdog Only with 1 Replica
Watchdog Only with 2 Replicas
Watchdog Only with 3 Replicas


Mean Time to Recovery (seconds)
3.000
2.000
1.000
0.000
0.188
0.289
0.232
0.278
0.227
0.380
0.262
0.760
0.420
0.700
2.642
0.200
0.214
0.565
0.183
0.225
1.175
0.241
All Defenses with 1 Replica
All Defenses with 2 Replicas
All Defenses with 3 Replicas
Watchdog Only with 1 Replica
Watchdog Only with 2 Replicas
Watchdog Only with 3 Replicas


Worst Time to Recovery (seconds)
20.000
15.000
10.000
5.000
0.000
0.400
1.050
0.550
1.550
0.500
1.750
0.650
1.750
2.050
2.300
0.350
13.950
0.450
2.100
0.250
0.450
0.500
15.400
All Defenses with 1 Replica
All Defenses with 2 Replicas
All Defenses with 3 Replicas
Watchdog Only with 1 Replica
Watchdog Only with 2 Replicas
Watchdog Only with 3 Replicas

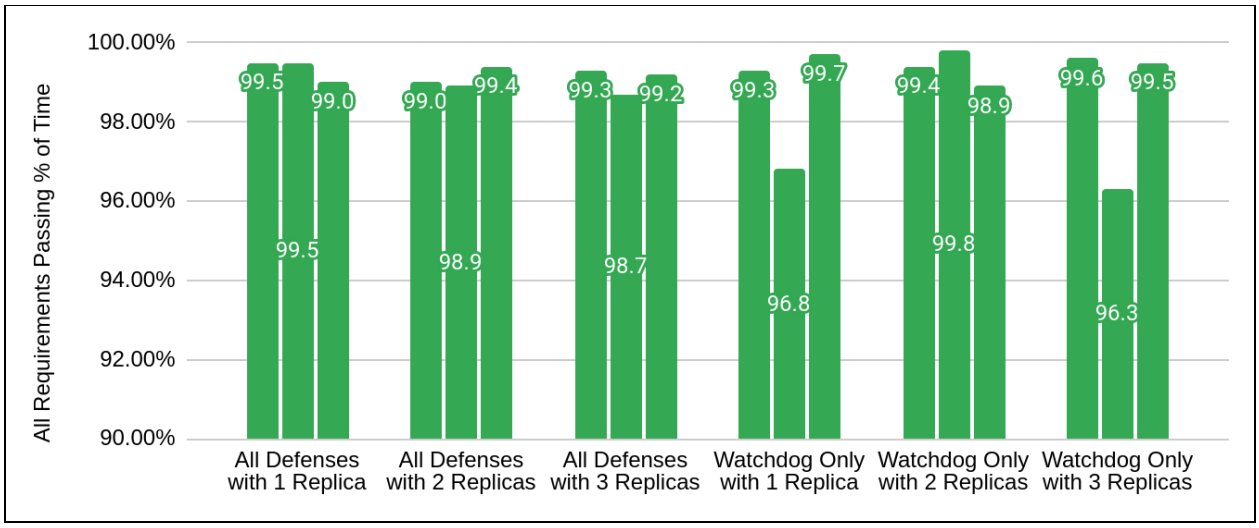


Figure 9-10: Register Fault Injection Results

The register error results parallel the memory error results. In the register trial results in Figure 9-10, the All Defenses with 1 Replica configuration once again outperformed all other configurations in fault recovery, in terms of having the lowest Worst Time to Recovery of 1.05 seconds. Among Watchdog Only results, the 1 Replica configuration again outperformed the other configurations in Mean Time to Failure; the Worst Time to Failure results were inconclusive. Once again, the data indicates that some of the defenses may be sabotaging the reliability improvements achieved by other defenses.

Determining with more confidence which defenses are successful and which defenses are counterproductive will require additional testing to track down the contributions of each feature in more detail.

10. Future Work and Summary

The future work for this research is broken down into two sections: first, addressing the limitations of the existing design and evaluation approaches (Section 10.1. Limitations); and second, experimenting with further techniques to defend Vivid applications against radiation errors and make the operating system more useful (Section 10.2. Extensions). To finish, section 10.3. Summary will provide a summary of the content of this thesis.

10.1. Limitations

The results from Section 9. Evaluation indicate that Vivid is effective, but it is unclear which of its techniques are important. The variation in results between different trials with the same configuration indicate a high probability that some of the results may be an artifact of randomness, not a true reflection of Vivid and the flight software's reliability. Future experiments should run for longer periods of time to produce more accurate averages.

The evidence from the evaluation indicates that some of Vivid's defenses appear to be counterproductive, because otherwise the three-replica configuration should have unequivocally performed better than the one-replica configuration. This difference, however, may also have been the result of shifting the *forms* of the failures observed; Hailburst currently does not distinguish between different forms of failure, such as silent data corruptions, partial component failures, and complete system failures. Distinguishing between these types of failure will help explain the impact of the different defenses. Future experiments should also evaluate each defense separately, rather than all at once. Future work on Vivid should address the deficiencies of the current defenses, or, if they are invariably counterproductive, remove them.

The current testing approach is an "all at once" approach; instead of injecting faults into specific components of the flight software, Hailburst injects faults randomly across all memory. While end-to-end testing is valuable for understanding the performance of the overall system, future experiments should include injections into specific components of the flight software, to allow identification of particular points of vulnerability.

Hailburst currently cannot inject faults into all pieces of the simulation state. QEMU does not allow GDB to modify values in all registers, and Hailburst modifies only main memory, not the memory mapped to particular devices. Future versions of Hailburst should support injecting faults into *all* components of the flight compute unit, not just main memory and a subset of registers.

Swivel is significantly simpler than a real-world spacecraft. Even if Vivid performs well on Swivel's current flight software, that does not mean it will perform well on full-scale flight software with more intensive and varied requirements. Further, the small scale of Swivel's flight software means that the implementation does not have to be efficient to meet its deadlines; the overhead of redundant multithreading is likely to be significant, and may make Vivid prohibitively inefficient for real projects. Future versions of Swivel should therefore include additional devices and flight software modules that behave in different ways (such as a simplified attitude control system). The performance of Vivid on extended flight software will better reflect whether it is feasible for real-world deployment.

10.2. Extensions

Beyond addressing the limitations of Vivid's current defenses and the existing testing protocols, future work may include expanding Vivid's feature set, both to add additional defenses and to expand the capabilities of flight software implemented on Vivid.

The current design of Vivid targets critical flight control software, but not all software requires the same level of criticality. Future versions of Vivid might include explicit support for mixed-criticality systems, where an engineer can deploy some components of the application with triple-redundancy, some with double-redundancy, and some without any redundancy. Further, instead of "waiting out" scheduling periods when a clip ends early, Vivid might switch to one or more clips performing scientific computing, such as data analysis, to spend extra time on the schedule productively.

One critical feature missing from Vivid's current design is memory isolation. Currently, a malfunctioning component can overwrite memory belonging to another component, which means that a single malfunctioning replica of a component might overwrite the data of the two other replicas for that component, and compromise Vivid's ability to seamlessly mitigate radiation errors. Some of the counterintuitive results seen in Section 9. Evaluation might be the result of this flaw, so future versions of Vivid should incorporate static configuration for memory isolation, in addition to the other features of the static component system.

While prepare/commit drivers may help Vivid avoid silent data corruptions, they are more vulnerable to crashes; if either replica fails, the driver will be unable to send or receive any data until it recovers. Because many of these failures result from code corruption, future versions of Vivid might include *dual* code replication for these clips. In this extension, rather than each clip having one copy of its code, each clip will have two copies of its code, and when it crashes,

Vivid will automatically switch it to use the other copy, to give the memory scrubber a chance to repair the first copy, but allow the driver to recover more seamlessly.

Vivid's requirement to calibrate clip durations based on *all* possible code execution paths is prohibitively difficult, because the durations of clips can vary based on how easily the input ducts computed their votes over the correct message. If all three inputs agree on a message, voting duct inputs are fast. If one of the three disagrees, it can take longer for the voting duct to decide which is the majority, because it has to perform more comparisons. This means that an incorrect clip duration calibration in a downstream component, combined with a single-replica failure in an upstream component, might cause a deadline overrun in all of the clips in the downstream component simultaneously. Future versions of Vivid will need to address this, either with constant-time voting ducts, or by providing better tools to calibrate clip durations, such as through code analysis.

Because it will always be possible for radiation to corrupt the kernel in ways that require watchdog intervention, the possibility of radiation-induced reboots is a fact of life under Vivid. Future versions of Vivid might want to streamline reboots further: particularly, by recovering application state after a reboot. It may be possible for the bootloader to skip overwriting read-write memory segments when it loads the kernel ELF file; this would allow Vivid to seamlessly restore the state of all application components, and, in effect, reboot the kernel without rebooting the flight software running on the kernel. As a result, the watchdog device could force a reset after a much shorter delay (such as a single scheduling cycle) without raising the risk of data loss, and significantly reduce Vivid's worst-case recovery time.

Finally, real flight software might consider using some of Vivid's design techniques, even if it runs on radiation-hardened flight computers. Vivid's static component system, as an

example, significantly reduces the amount of runtime code and time required to initialize flight software by eliminating the need to perform runtime resource allocation. Because some real-world flight software is highly sensitive to reboot times, using this technique may help the software recover faster from reboots that occur for other reasons besides radiation.

10.3. Summary

This Master of Engineering thesis addressed the challenge of defending flight software from radiation errors without running the software on a radiation-hardened processor. It introduced Vivid, a hard real-time operating system that allows engineers to develop redundantly-multithreaded flight software using a set of radiation-resistant abstractions and a compartmentalized operating system structure. It presented a hypothetical testbench spacecraft, Swivel, and defined a list of behavior requirements for Swivel's flight software to satisfy. It demonstrated the practical use of Vivid by writing SwivelFSW, an implementation of flight software that satisfies Swivel's requirements, and described the SwivelSim simulation for the spacecraft's avionics, which included simulating the processor that runs the flight software. This thesis also introduced Hailburst, an efficient and deterministic fault injection tool, and used Hailburst and SwivelSim to evaluate the ability of the SwivelFSW flight software running on Vivid to tolerate radiation faults. The system seamlessly tolerated approximately 149 out of every 150 radiation faults injected by Hailburst, with no observed requirement failures, and recovered from the remaining 1 out of 150 radiation faults within at most 2.05 seconds in the worst observed case. Some of Vivid's defenses appear to be more effective than others, and some may be counterproductive, so future work is required before engineers can consider applying Vivid's design principles to real-world flight software.

Bibliography

[1] J. Eickhoff, *Onboard Computers Onboard Software and Satellite Operations - An Introduction*, Heidelberg, Germany: Springer-Verlag, 2012. [Online]. Available: https://link.springer.com/book/10.1007%2F978-3-642-25170-2. Accessed: May 10, 2021.

[2] "The Mars 2020 Rover's 'Brains'," NASA JPL. https://mars.nasa.gov/mars2020/spacecraft/rover/brains/ (accessed May 15, 2022).

[3] K. A. LaBel, A. H. Johnston, J. L. Barth, R. A. Reed and C. E. Barnes, "Emerging radiation hardness assurance (RHA) issues: a NASA approach for space flight programs," *IEEE Transactions on Nuclear Science*, vol. 45, no. 6, pp. 2727-2736, Dec. 1998, doi: 10.1109/23.736521.

[4] P. Mehlitz and J. Penix, "Expecting the Unexpected - Radiation Hardened Software," AIAA 2005-7088. *Infotech@Aerospace*. Sep. 2005, doi: 10.2514/6.2005-7088.

[5] A. G. Schmidt, M. French and T. Flatley, "Radiation hardening by software techniques on FPGAs: Flight experiment evaluation and results," *2017 IEEE Aerospace Conference*, Big Sky, MT, 2017, pp. 1-8, doi: 10.1109/AERO.2017.7943651.

[6] R. DeCoursey, R. Melton, and R. Estes, "Non radiation hardened microprocessors in space-based remote sensing systems", *Proc. SPIE Sensors, Systems, and Next-Generation Satellites*, vol. 6361, 2006.

[7] B. Döbel. *Operating System Support for Redundant Multithreading.* Dissertation, Technische Universität Dresden, 2014.

[8] H. Leppinen, A. Kestilä, P. Pihajoki, J. Jokelainen and T. Haunia, "On-board data handling for ambitious nanosatellite missions using automotive-grade lockstep

microcontrollers", *Small Satellites Systems and Services - The 4S Symposium 2014*, May 2014.

[9] "NASA Orion," Green Hills Software. https://www.ghs.com/customers/nasa_orion.html (accessed May 15, 2022).

[10] "Radiation-hardened electronics product guide," BAE Systems. https://www.baesystems.com/en-media/uploadFile/20211206201500/1434554723601.pdf (accessed May 15, 2022).

[11] M. N. Lovellette et al., "Strategies for fault-tolerant, space-based computing: Lessons learned from the ARGOS testbed," *Proc. IEEE Aerospace Conference*, Big Sky, MT, USA, 2002, pp. 5-5, doi: 10.1109/AERO.2002.1035377.

[12] R. Vemu and J. A. Abraham, "Ceda: control-flow error detection through assertions," in *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, July 2006.

[13] N. Psenjen et al., "Toward a Suite of Middleware Services for Enhanced Spacecraft Configuration and Capability." *Spacecraft Flight Software Workshop (FSW 2015)*, Johns Hopkins APL, Laurel, MD, Oct. 27-29, 2015.

[14] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annual Technical Conference*, 2014, pp. 305–320.

[15] J. Rufino, J. Craveiro, and Verissimo, P., "Building a time- and space-partitioned architecture for the next generation of space vehicle avionics." In *Software Technologies for Embedded and Ubiquitous Systems: 8th IFIP WG 10.2 International Workshop*, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Berlin: Springer Berlin Heidelberg, 2010, pp. 179-190, doi: 10.1007/978-3-642-16256-5_18.

[16] K. Boos, N. Liyanage, R. Ijaz and L. Zhong, “Theseus: an Experiment in Operating System Structure and State Management.” *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2020.

[17] H. M. Quinn, D. A. Black, W. H. Robinson and S. P. Buchner, "Fault Simulation and Emulation Tools to Augment Radiation-Hardness Assurance Testing." *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 2119-2142, June 2013, doi: 10.1109/TNS.2013.2259503.

[18] D. Ferraretto and G. Pravadelli, "Simulation-based fault injection with QEMU for speeding-up dependability analysis of embedded software." *Journal of Electronic Testing*, vol. 32, no. 1, pp. 43-57, Jan 2016.

[19] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator.” *USENIX Annual Technical Conference*, pp. 41–46, 2005.

[20] E. Carlisle IV and A. George, “Dynamic robust single-event upset simulator.” *Journal of Aerospace Information Systems*, 15(5), pp. 282-296, 2018.

[21] K. Wolf et al., “docs/interop/qcow2.txt,” QEMU Git. https://git.qemu.org/?p=qemu.git;a=blob;f=docs/interop/qcow2.txt;hb=c313e52e6459de2e9064767083a0c949c476e32b#l690 (accessed May 18, 2021).

[22] J. An, H. You, F. Xie, Y. Yang and J. Sun, "FIG-QEMU: A Fault Inject Platform Supporting Full System Simulation." *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*, Xi'an, China, 2020, pp. 275-278, doi: 10.1109/DSA51864.2020.00049.

[23] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, “FAIL*: An open and versatile fault-injection framework for the assessment of

software-implemented hardware fault tolerance.” *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, pages 245–255. IEEE Computer Society Press, Sep. 2015.

[24] J. Ganssle, “Great Watchdog Timers For Embedded Systems,” *The Ganssle Group*, Sep. 2011. http://www.ganssle.com/watchdogs.htm (accessed May 15, 2022).

[25] “SCons 4.0.1,” *The SCons Foundation*, 2020. https://scons.org/doc/4.0.1/HTML/scons-user.html (accessed May 15, 2022).

[26] “Embedded Artistry libc,” *Embedded Artistry*. https://embeddedartistry.github.io/libc/index.html (accessed May 15, 2022).

[27] J. Gailly and M. Adler, “A Massively Spiffy Yet Delicately Unobtrusive Compression Library,” *zlib Home Site*. https://www.zlib.net/ (accessed May 15, 2022).

[28] “Using the GNU Compiler Collection (GCC),” *The GNU Compiler Collection*. https://gcc.gnu.org/onlinedocs/gcc-8.3.0/gcc/ (accessed May 15, 2022).

[29] M. S. Tsirkin and C. Huck, “Virtual I/O Device (VIRTIO) Version 1.1,” *OASIS Committee*, Apr. 11, 2019. https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.html (accessed Aug 22, 2021).

[30] S. Parkes, *SpaceWire User’s Guide*. Dundee, Scotland, United Kingdom: STAR-Dundee Ltd, 2012. [Online]. Available: https://www.star-dundee.com/wp-content/star_uploads/general/SpaceWire-Users-Guide.pdf. Accessed: May 10, 2021.

[31] “Remote memory access protocol (normative),” *European Space Agency*, 2006. http://spacewire.esa.int/content/Standard/documents/SpaceWire%20RMAP%20Protocol%20Draft%20F%204th%20Dec%202006.pdf (accessed Dec 7, 2021).

[32] “The Go Programming Language,” *Google*. https://go.dev/ (accessed May 15, 2022).

[33] G. D. Forney, "The viterbi algorithm," in *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268-278, March 1973, doi: 10.1109/PROC.1973.9030.

[34] D. White, “Considerations Surrounding Single Event Effects in FPGAs, ASICs, and Processors,” Xilinx White Paper, March 7, 2012.