SpaceWhale: A Space-based Fault-injection Framework

Daniel Krolopp and Aaron Neustedter
Purdue University, IN, USA
{dkrolopp, aneusted}@purdue.edu

Abstract

With an increased eye towards 21st-century space exploration, a need has arisen for frameworks to simulate realistic faults in space-based distributed computer systems. We propose a new testing framework called SpaceWhale, based off the open-source, minimal Xinu operating system. SpaceWhale serves as a platform for simulating bit flips in different segments of kernel and application memory.

Throughout this paper, we describe two sample distributed systems and test their resilience against memory faults often found in space applications using SpaceWhale. The first configuration is a simple linear chain, whereas the second is 3-way voting scheme consisting of four individual nodes. Through our framework, we experimentally demonstrate that the linear configuration is superior to the voting mechanism for our chosen fault rate and sample workload.

I. Introduction

In the past two decades, interest in space exploration by the public and private corporations has increased dramatically. In the past 10 years, private vehicles such as Blue Origin's New Shepard, SpaceX's Falcon 9 Booster and Dragon capsule and The Spaceship Company's SpaceShipTwo space plane have made huge advancements in reusability and significantly decreased launch costs. It now appears that private corporations are leading the charge in aerospace innovation. Unlike government space operations, these companies are many times extremely cost conscious and often do not have the money to spend on expensive radiation-hardened equipment (The IBM RAD6000 and RAD750, best known for their use on NASA spacecraft reportedly costs over $200,000 per unit [3]). One alternative is to use commercial off the shelf equipment and implement the required redundancy in software.

We seek to simulate some of the effects of radiation on an ecosystem of devices and explore the effects it has on a computation that these devices are working on together.

II. Idea and Approach

Assumptions

Accurately simulating the environment that a computer system would experience in the vacuum of space is an enormous challenge and out of scope for this project. In order to focus on the software simulation aspect, we made several large simplifying assumptions on the faults that would occur. We choose to focus only on "Single Event Transient" errors, in other words, single bit flips in memory [1]. In addition, we assumed occurrences of these faults occur in a random fashion uniformly distributed in terms of time of occurrence and location in memory.

Because of these assumptions, this work could be generalized for computers operating in non-space environments that still encounter frequent memory errors.

Test Stand Requirements

We determined that for a test stand to meet our goals, it had to meet the following requirements:

- Ease of repeated deployment and termination of multiple test machines
- Ability to inject memory faults into all areas of memory (kernel or user)
- Easy centralized deployment and result collection
- Scalability
- Potential for automation

Our implementation does not meet all of these requirements perfectly, but we believe that it is the best choice for the job based on experience from other approaches that we attempted

Test Stand Design

Before settling on our Xinu based approach, we attempted to create completely virtualized test stand using libvirt and virsh to manage several qemu instances of Ubuntu Server Edition 18.04. We felt that the portability and lack of hardware requirements of such an approach would be advantageous, however issues arose in injecting faults in such an environment. All modern-day operating systems have difficult to circumvent protections that do not allow modification of kernel memory, and triggering memory faults using an external framework was not easily accomplished with current versions of all common virtualization software. As fault injection is central to our test stand, we could not use this approach

There are a number of advantages to using Xinu for our approach.

Xinu is a small, cross-platform operating system, designed for use in both academic and industrial purposes. [2] Xinu has seen most of its use in embedded systems, ranging from printers to automobiles to pinball machines. For this reason, we found it a good fit for approximating embedded systems found in space-based contexts.

Additionally, Xinu runs entirely in privileged CPU mode, so all applications running on the operating system can view and modify any region of memory. We take advantage of this fact to help us build our fault injection framework. Our bundled version of xinu includes sample applications, as well as a process that modifies memory to simulate the effects of cosmic radiation.

Finally, Xinu is an extremely minimal operating system, and as such has an incredibly small memory footprint. With larger operating systems, large portions of kernel memory are not accessed during the lifetime of a single, simplistic application (such as the one we're running). With Xinu, kernel memory weighs in at just under 60 KB, complete with our sample workload applications. This is advantageou since any memory faults we inject are likely to impact code or data critical to our system's operation and produce meaningful results.

Our specific Xinu implementation is the ARM variant, running on Texas Instruments' BeagleBone Black SoCs that communicate with each other using UDP internet protocol. For our distributed approach, we used Purdue University's Xinu lab which gives us access to 185 Beagle Bone Blacks, all of which we can load our code onto and launch remotely from a central backend server. In this work we use a maximum of four machines at one time, but these extra machines give us headroom to create more complex configurations and/or parallelize testing in the future.

Sample Workload

Our sample workload is an application we named *CPU-bound*. The main idea is to create a 4 KB "workspace" that stores integers which we repeatedly sum to produce a final value. This serves as an amply-large target for injected faults to land upon and disrupt normal computation. The *CPU-bound* algorithm is as follows:

```
/* input is in the form of
 * iters: # of iterations to perform
 * input: seed input (int)
 */

wkspace[1024];
for i = 1 -> 1024:
  wkspace[i] = i * input ^ 2

for i = 0 -> iters:
  for j = 1 -> 1024:
    for k = 1 -> j:
      wkspace[j] *= wkspace[k] + 1

return wkspace[1023]
```

Figure 1: The *CPU-bound algorithm.*

This algorithm is designed such that if a fault occurs in our *wkspace*, it is highly likely to result in our final computation being incorrect. The only case in which it could recover is in the rare event that another fault occurs in such a way that it restores our workspace to an uncorrupted state.

Fault Injection

In order to inject memory faults, we had an additional process running in the background of each machine that slept for a uniformly random amount of time between 0 and 2 seconds. Upon waking, it would choose a random bit in memory, record which section of memory it was in, flip it, and return to sleep. This would continue until the sample workload was finished or the system crashed due to the accumulation of errors. The average of 1 fault per second used is unusually large rate of fault occurrences, this was done to simulate a longer period of lower fault rate in a shorter period of testing.

Testbed Configuration

We chose two main implementations for fault-tolerance analysis. The first of which was designated the *Linear configuration*, and consisted of three interconnected nodes each with the same *CPU-bound* workload. In this case, the user running the system designated *iters* and *input* for the first node, and *iters* for the second and third nodes. The resulting value from the first *CPU-bound* node is fed into the second node as *input*, and the same relationship holds for the second and third node.

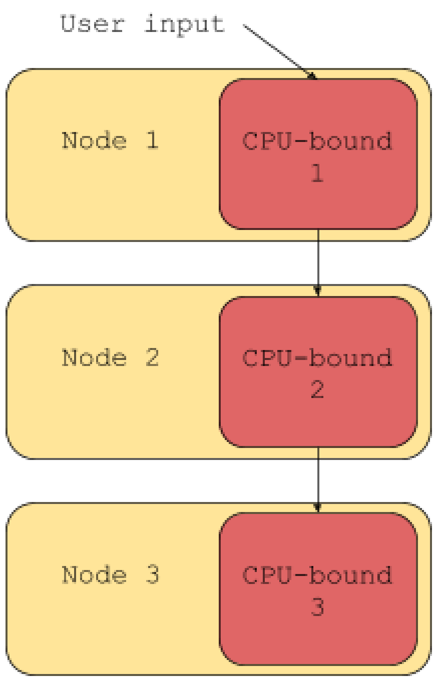

Figure 2: The *Linear configuration* setup. User input in the form of a seed value is passed to the first node. Each successive node receives input from the previous.

The second implementation we denoted the *Voting scheme*. This consisted of three nodes, each running three rounds of *CPU-bound* individually, before passing their results along to an arbitration or voting node. This voting node then determines whether a majority can be established based on the results received, and if so, establishes that as the result of the three rounds of *CPU-bound*.

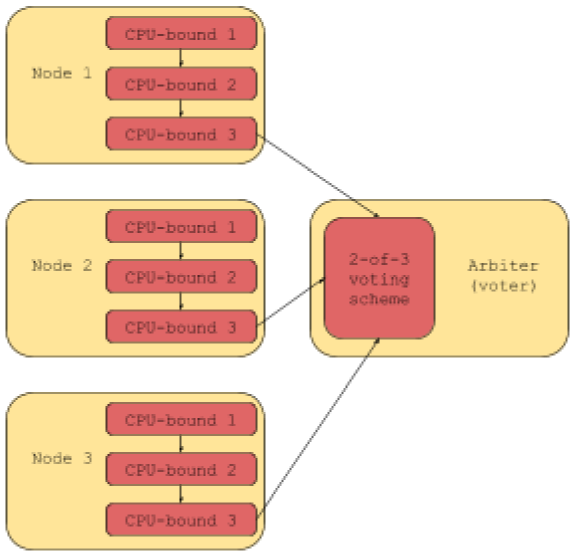

Figure 3: The *Voting scheme* setup. User input in the form of a seed value is passed to each node. Each node performs three rounds of *CPU-bound*, passing the output of one round in as the input to the next. Results are passed to an arbitration (voter) node, which attempts to determine the correct value.

Data Collection

We design our investigation to allow us to collect data on what types of errors our space-based memory faults can generate. As auxiliary motivations, we want to measure the frequency with which memory corruptions occur, and where in memory they happen.

Every time the fault injection process triggers, it updates a mapping of memory segments to a count of bit flips that have occured in that segment. Once each node completes its workload, we display the mappings.

With regard to the faults observed, human interaction is required. Bit flips in kernel memory can cause kernel panics, which rely upon the experimenters to record and reboot the system. Additionally, we rely upon human verification that the computation produces the correct result.

III. Experimental Methods

Golden Runs and Execution Profiling

Our observations of both systems begin in the same way, with golden runs consisting of fault injection rates of 0.0. This establishes expected values our system should output during runs involving fault injection and time frames during which we can expect the system to finish its workload.

In both cases, we seed the systems with a user *input* of 314, *iters* of 100 and expect the decimal value 1700241152 as a result. Based on our execution profiling, we expect each round of *CPU-bound* to take approximately 25s to complete on our hardware.

Experiment Instrumentation

For our *Linear configuration* setup, we run a sample experiment 15 times. For our *Voting scheme*, we run the experiment 11 times.

In both configurations, for both experiments, we seed our rand functions with a unique value in the form of the current system time. Both run with a global fault injection rate averaging one bit flip per node per second.

Since the experiments run on a distributed system, we boot each node at approximately the same time, allowing each to accumulate approximately the same number of faults within the same time periods. After each experiment, the machines are power-cycled.

We record each experiment as either a success or a failure. Failures are defined as any (or a combination) of the following:

1. A node crashes
2. The system arrives at an incorrect computed value
3. A Byzantine failure occurs

When a failure occurs, we record the type of failure and in which node it occured.

IV. Results

Linear Configuration

Figures 3 and 4 below show the distribution of faults during testing. Figure 3 shows that the number of faults injected increases in a linear fashion from node 1 to node 3. This makes sense as faults are no longer injected after computation is complete, so node 2 and 3 accumulate faults waiting for node 1 to finish it's computation. Assuming that each node runs for approximately the same amount of time, the linear nature proves that the faults are being injected with uniform randomness.

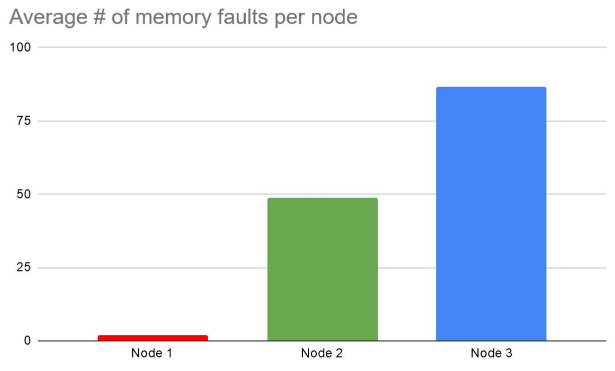

Figure 4: Average number of memory faults injected in each node at the end of it's sample workload computation

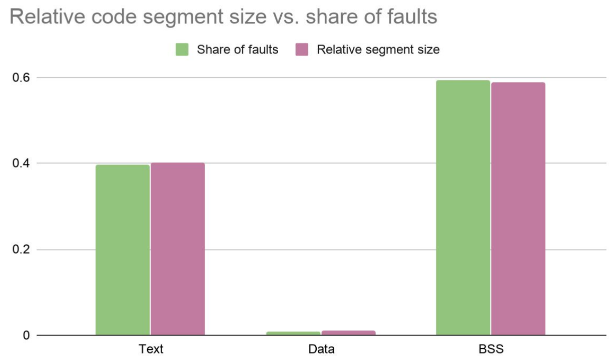

Figure 5: Code segment size compared to the number of faults occuring in each segment

As shown in Figure 3 below, the linear configuration setup produced a successful result 35.7% of the time. Outside of these successes, we saw failures due to incorrect computations most of the time, by crashes and byzantine failures. Although a 35% success rate is fairly low, when considering that an exceptionally minimalistic operating system with no redundancy is was able to operate without error 30% of the time when over 80 bytes of memory were corrupted is quite interesting. The chance of a successful operation could be increased by decreasing the complexity of the example workload or lowering the fault injection rate, and this should be done in future tests.

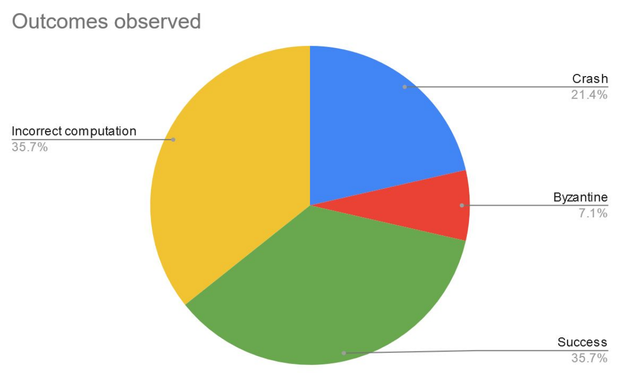

Figure 6: Ratio of test outcomes observed

In Figure 6, we see the distribution of nodes which caused failures during testing. When comparing this chart to Figure 3, it makes sense. Node 1 never failed, but it was often operating with a significantly lower amount of faults as node 2 and 3. Node 2 failed more often that node 3 because once node 2 failed, node 3 no longer had a chance to fail

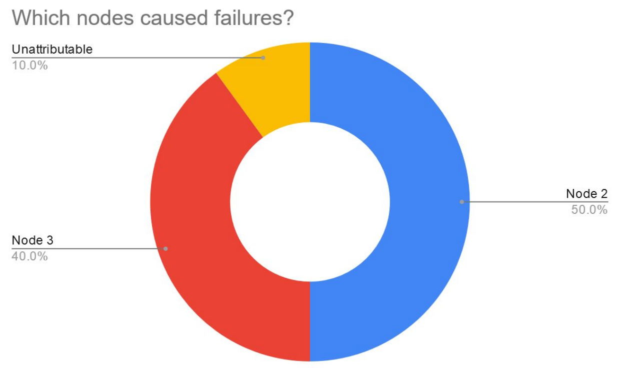

Figure 7: Ratio of nodes that caused the system failure. Unattributable errors where any one node is not at fault (communication errors, etc)

Overall, the results from our linear configuration show little surprise and provide a good baseline for comparison to other configurations, one such is the voting scheme we implemented, discussed below.

Voting Scheme

During our testing of the voting configuration, we did not observe a single successful run. While surprising to the casual observer, we noted that our system became even less-tolerant of memory faults than our *Linear configuration.* When we examine the math, the explanation is clear:

$$\binom{3}{2}(\frac{1}{3})^2 \cdot (1-\frac{1}{3}) \cdot (\frac{1}{3}) = 0.074$$

Note that our probability of success is taken from the experimental value derived from the experiment of the first set. In this configuration, each node is assumed to have a probability of success equal to ⅓. With a voting scheme, at least two nodes must make the correct computation for the overall system to be correct. Additionally, we have a fourth node of the arbiter/voter which risks faults itself. This reliance upon numerous nodes all working flawlessly explains our experimental results.

Additionally, since each node that participates in the system must be alive long enough to submit its result to the voting node, more compute-time is used, allowing more time for faults to accrue.

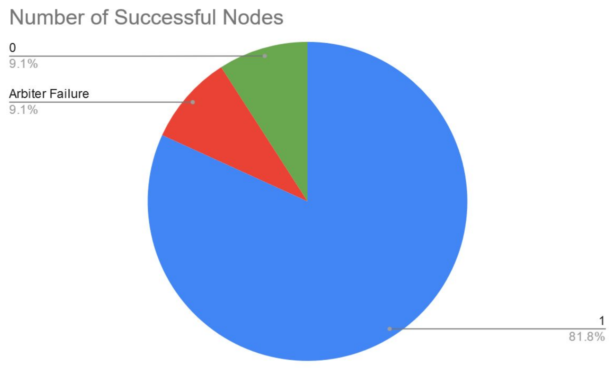

Figure 8: Number of nodes that calculated the result successfully. Note that 2 successful nodes in one run never occurred, the requirement for a successful test

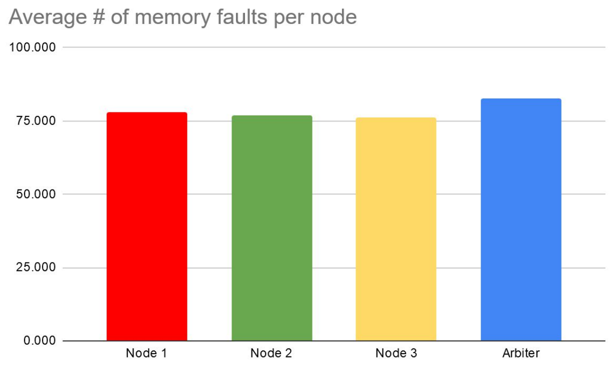

Figure 9: Average number of memory faults injected per node

In figure 10 below, we see the ratio of success to failure and the breakdown of failure modes. This is very similar to the linear configuration we saw earlier, likely due to the identical workload and memory layout

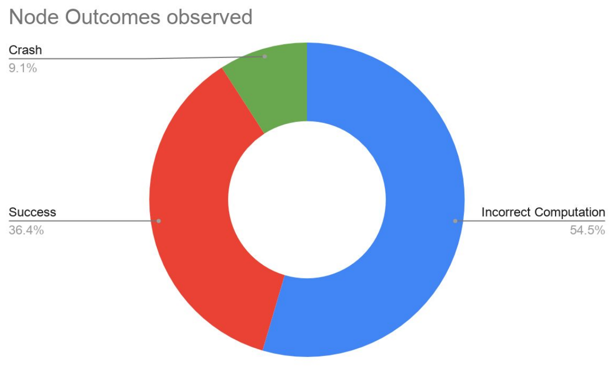

Figure 10: Outcome distribution for each individual node.

In figure 11 below, we see the number of successful completions per node. Although the Arbiter typically accumulated the most faults, it's high success rate is unsurprising as it has a relatively simple job of comparing 3 numbers and then terminating. However, we would expect the other 3 nodes to have similar computation success rates, which is not the case. The exact cause of this in undetermined, but we suspect that the nodes were powered off for enough time between tests and some of the RAM was not successfully cleared. This warrants further investigation

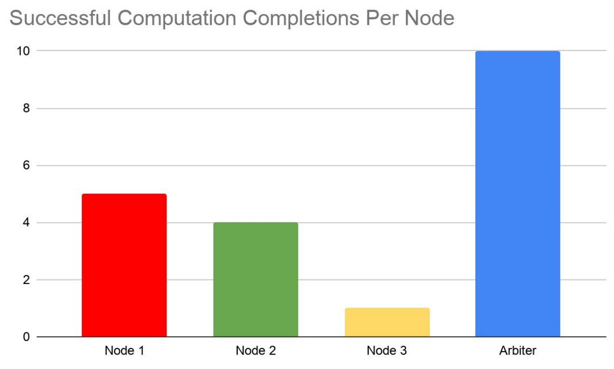

Figure 11: Number of successful computations per node

V. Conclusions and Future Work

Shortcomings

Due to the time-intensive data collection method we employed, the amount of data we were able to collect was limited, which caused uncertainties in our analysis. We have identified many areas of the data collection that could be automated, and investment of time to improve in this area would allow for more data to be collected for less effort.

We only tested one fault injection rate and two configurations of our test stand. This only gives us a small peek into the behavior of such an environment. Re-execution of our tests with many different fault injection rates and different configurations (Series-parallel systems would be of particular interest) would lead to more complete and interesting data analysis. Improvements in the above-mentioned automation would make this task significantly faster and easier.

In addition to the one injection rate and two configurations, we only tested one sample workload, a workload that cannot recover for any bit errors. This is not representative of all programs, and an additional workload that can tolerate a limited amount of bit flips would be interesting. This type of workload would be representative of many machine learning based applications, an application that is growing in popularity.

Conclusions

In addition to our contribution of a teststand and procedure able to simulate an ecosystem of devices operating in a hostile environment, we also ended with several takeaways from our limited testing.

Fault-tolerant distributed system design takes a lot of thought to get right. Assuming that a voting system or other “typical” redundancy methods will improve the system reliability without first considering the consequences and real-world environment could easily leave the system in a more-vulnerable state. Sufficient testing of any additional fault tolerant features is necessary to ensure that the system as a whole operates as expected. This is especially important in situations like those found in space applications, where a single software failure can cause the failure of a spacecraft and cost millions of dollars and endanger the lives of those onboard

References

[1] E. Normand and L. Dominik, "Cross Comparison Guide for Results of Neutron SEE Testing of Microelectronics Applicable to Avionics," *2010 IEEE Radiation Effects Data Workshop*, Denver, CO, 2010, pp. 8-8. doi: 10.1109/REDW.2010.5619496

[2] The Xinu Lab at Purdue https://xinu.cs.purdue.edu/#lab

[3] “BAE Systems moves into third generation rad-hard processors". Military & Aerospace Electronics. 2002-05-01