Detection and Prevention of Silent Data Corruption in an Exabyte-scale Database System

David F. Bacon
Google
New York, New York, U.S.A.
dfb@google.com

***Abstract*—Google's Spanner database serves multiple exabytes of data at well over a billion queries per second, distributed over a significant fraction of Google's fleet. Silent data corruption events due to hardware error are detected/prevented by Spanner several times per week.**

For every detected error there are some number of undetected errors that in rare (but not black swan) events cause corruption either transiently for reads or durably for writes, potentially violating the most fundamental contract that a database system makes with its users: to store and retrieve data with absolute reliability and availability.

We describe the work we have done to detect and prevent silent data corruptions and (equally importantly) to remove faulty machines from the fleet. We present a simplified analytic model of corruption that provides some insights into the most effective ways to prevent end-user corruption events.

We have made qualitative gains in detection and prevention of SDC events, but quantitative analysis remains difficult. We discuss various potential trajectories in hardware (un)reliability and how they will affect our ability to build reliable database systems on commodity hardware.

***Index Terms*—databases, corruption, hardware faults**

I. Introduction

Spanner [1], [2] is Google's planet-scale database that supports 5 of Google's 7 products with a billion or more users. Other key infrastructure on Spanner includes AdWords, the Zanzibar access control system [5], and much of the control plane for Google Cloud. In aggregate Spanner serves over 1 billion queries per second on multiple exabytes of data stored in tens of thousands of databases, and is one of the largest consumers of resources in Google's fleet.

Availability and data integrity are therefore fundamental to Spanner.

Spanner data is generally replicated in 3 or more geographically distributed data centers to provide fast fail-over in the event of failure, remaining robust even if all data centers in a metro region are offline. These replicas can also be used to detect and repair data corruptions. Some customers, such as Zanzibar, also add tens of read-only replicas to provide even lower and more consistent latency for reads across geographies.

Spanner's availability SLA is 99.999% ("five nines"). Because of its stringent correctness requirements and its scale, Spanner has been at the forefront of detecting the phenomenon of Silent Data Corruption [3], [4] within Google, and in implementing measures to detect and prevent it. Spanner's geo-replicated nature also means that we can more frequently detect corruptions than unreplicated services.

In addition to scale, Spanner also has a much higher workload diversity than other systems of comparable scale (such as the Colossus file system). There is a diversity of database schemas, of read/write workloads, of concurrency semantics, and of simple reads and writes versus highly complex SQL sub-programs. This creates a much bigger "attack surface" for bugs.

Appeared in the IEEE conference on Silicon Errors in Logic — System Effects, May 19, 2022 (held virtually)

II. Detection

Because of their rarity and diverse manifestations, reliable detection of SDC events is a fundamental challenge.

A. Crash Triage

Every crash of a Spanner server in production is investigated by an engineer, who is responsible for identifying clusters of failures, filing bugs, and routing them to the appropriate engineers for root-causing and fixing. They also coordinate with the Site Reliability Engineering team that runs the Spanner service in the event that a bug is detected that warrants production intervention. Such interventions include rolling back a software release, disabling a new feature, isolating a problematic tablet, or otherwise mitigating the issue.

Crash triage is essential for maintaining Spanner's quality. Despite running a huge battery of tests on every production release (Spanner is Google's single largest consumer of testing resources), as with all systems bugs do make it past testing. And Spanner's scale means that even extremely rare bugs are often exercised by some workload.

B. Detecting Silent Data Corruption in Production

Almost all software bugs exhibit some sort of correlation. Most straightforwardly, a failing invariant check or a null pointer dereference always generates the same stack trace. But even bugs that are difficult to root-cause, like use-after-free, usually exhibit some form of correlation. For instance, the corruptions might all occur in objects of the same size class as the prematurely freed object.

A number of years ago, Spanner's crash triage team began to observe an increase in crashes whose only correlation was that they occurred on the same machine. Over time we realized

that the trend was worsening, and the increase couldn't just be ascribed to an increase in the size of the fleet or in Spanner's increased footprint within the fleet.

Over time we found that the following symptoms, individually or in combination, were reliable indicators of bad machines:

- An elevated level of kernel panics.
- An elevated level of SIGSEGV, SIGILL, and SIGFPE crashes in Spanner or other "high reliability" binaries, especially when concentrated on a single core or hyperthread pair.
- An elevated level of unrelated fail-stop invariant check failures.
- Any fail-stop checksum mismatch failures. Such failures are particularly strong indicators.

We also learned that such events could be widely separated in time, sometimes by over a year, and still provide a meaningful diagnostic signal. For example, a Spanner memtable checksum failure plus a 9-month old kernel panic may be sufficient to mark the machine as bad.

As we gained experience, and became more confident of our reading of the symptoms, and as we socialized the problem with the platforms team responsible for the hardware, we were able to diagnose faulty machines more and more quickly. The capability to remove machines from service was also delegated to triage engineers.

Because of the non-deterministic and ultra-low frequency of these failures, the normal procedure for "fixing" broken machines had to be changed since normal screening and remediation procedures often failed to address the issue, and bad machines were returned to the fleet. We called these machines "recidivists".

Over time, procedures were put in place to explicitly eject corruptors so that they were not simply put back in production. Programs specifically designed to screen for bad machines were developed internally and with CPU vendors. These screens can catch a significant number, but by no means all, bad machines.

1) Validating SDC Events: How can we distinguish silent data corruption from disk and network errors? Disk and network are protected by hardware- and software-level checksums. In addition, Spanner's compaction logic computes an application-level checksum, then decodes, decrypts, and validates the checksum of each block before it is written. It also performs the same checksum computation as the underlying filesystem on the in-memory file contents, and uses a special API that returns the checksum of the file when it is closed. That checksum is validated against the in-memory file-level checksum.

How can we distinguish SDC from software failures? Software errors virtually always recur across multiple machines. In a few cases we have determined that multi-machine corruptions are hardware-induced, but the bar for such a conclusion is very high and needs to be supported by other evidence (e.g. all suspect machines failing a screening test).

How can we distinguish SDC from failures in other parts of the machine, e.g. in DRAM? SDC is almost always limited to a single core, or occasionally to a group of cores sharing a last-level cache. Once a machine is identified as likely to be faulty, we look for such core-concentrated crashes – both in the wild and in our growing repertoir of screening tests.

Note that we do not distinguish between corruption in on-chip caches versus on-chip functional units (ALU, FPU, etc). Operationally, it's irrelevant – in either case our only recourse is to replace the bad chip.

SDC signals can often be immediately isolated to a particular core, for instance when there are repeated segmentation faults. In other cases, like checksums on in-memory data structures, the detection code may not be proximate to the event where the corruption occurred. In such cases, we may be able to validate the corruption with an offline screening program. In other cases, we return the chip to the vendor who may be able to verify that the chip is bad.

In the end, determining whether a chip is bad remains part art, part science. The per-chip failure rate is so low and the failure modes sufficiently diverse that statistical analysis and modeling remain elusive. Over time as our screening processes and vendors' analysis improves, we are able to confirm past hypotheses. But new failure modes continue to manifest as well.

C. Audit

Since Spanner databases are geo-replicated, one mechanism for detecting and repairing corruptions is to compare the contents of the database across replicas. Spanner runs such an audit over each database, by default once per week. There are also consistency checks between indices and base tables, and structural invariant checks.

In the very rare event that a corruption is detected, the corrupted tablet in the minority is destroyed by an operator and automatically reconstituted from the other replicas. This mechanism has been helpful in repairing both hardware- and software-induced corruptions.

However, there are limitations: most obviously, the lag between corruption and detection could be as long as a week. Since they perform full scans of all replicas, the audits are very expensive.

More subtly, since Spanner is a multi-version database, compactions are not synchronized across replicas, and the implementation has flexibility in the representation of deletions, we can not compare structural equality. Instead we compare semantic equality at a chosen snapshot timestamp: that is, the sequence of user-visible rows yielded by a full scan of each table. As a result, a corruption at t_1 in a single replica could be briefly visible but then hidden as that data is overwritten at t_2. The database is still in a corrupted state, in the sense that a read below t_2 will see the corrupted data, but it will not be detected if the snapshot timestamp for the audit is at or above t_2.

D. Hardware/Software Bug Antagonism

An under-appreciated aspect of SDC is that it muddies the waters and slows down the detection and root-causing of both hardware and software bugs. The more untrusted components are in a system, the slower engineers will be to put effort into deeply investigating any particular component. Debugging corruptions is very hard, and after an engineer has spent hours or days chasing the cause of a corruption, only to conclude that it was due to a transient hardware error, they will be less likely to pursue future errors with such zeal until there are multiple occurrences.

A high level of SDC errors will materially slow down the rate at which critical software errors are detected, in some cases sufficiently that a bug isn't root-caused before hitting production. This isn't theoretical: we have suffered bugs in production because a very rare failure was mis-classified as SDC.

SDC therefore reduces software reliability in addition to hardware reliability.

III. Prevention

A. Checksumming

Checksumming has long been part of the arsenal for protecting data at rest and in transit. As the prevalence of SDC has grown, it has become an integral part of both our detection and prevention of SDC. Spanner already had significant checksum protection for its major in-memory data structures (the mutable data or "memtables" and the generation of immutable persisted layers of the LSM tree). These have prevented a significant number of SDC corruptions, as well as acting as one of our most reliable indicators of bad machines.

In response to SDC, we added checksum protection in more places (for instance, buffers used in less frequent code paths), and focused on end-to-end checksumming, so that data is never "naked" as it makes its way through various layers of the system. This ideal is never fully realizable, since many data transformations do not lend themselves to computation in a checksummed domain. Nevertheless, it is effective.

SDC can generate various types of incorrect computation, which includes incorrect addresses. Thus we can no longer rely on hardware protection of RAM against corruption (e.g. with ECC). To defend against dynamic corruptions caused by SDC, long-lived structures (e.g. in the database RAM block cache) are periodically re-checksummed. We also perform simple validation checks on the cache block header on each lookup.

Finally, we have added checksum protection to data structures that are small but have a particularly high blast radius. The most obvious example is encryption keys, but other examples include summary data that covers an entire LSM layer, such as the key range or timestamp range that it contains, since if those are corrupted, an entire layer of data could be erroneously omitted from a query.

B. Invariant Checks

Screening for bad machines and running audits are important tools for detecting SDC events, but they lack a fundamental desirable property: fail-stop behavior. Ideally, even if we couldn't prevent faulty execution, if we could crash the machine before incorrect results were written or returned, the system would be far more resilient.

However, the nature of SDC is that a significant number of errors are not fail-stop. In this case, the best we can do is to stop as early as possible. An effective approach is to add invariant checks.

Invariant checks in general are good software engineering practice, and can catch both hardware and software bugs. However, to be effective at finding SDC, invariant checks need to cover a large swath of computation and memory. For example, a traditional invariant check like "$a < b$" for the inputs to a function is unlikely to be effective. On the other hand, invariants such as checking that all key values returned in a scan are in sorted order, or all the timestamps in an append-only queue are increasing, have been effective in practice.

Such checks have the advantage that while they may not be fail-stop, they can halt the system before bad results can be returned.

Fail-stop checks work well for uncorrelated low-probability faults, but can cause crash-loops in the case of software bugs which do have correlations. Thus we are often trading off resilience to hardware corruptions versus software bugs – each invariant check is a judgment call.

C. Software Ejection of Bad Cores?

Our current practice is to eject a bad machine once we have sufficient evidence. An alternative would be to have the kernel simply not schedule any work on the bad core, analogously to what is done in flash chips for bad blocks.

However, bad cores can only be reliably identified for fail-stop failures. In some cases we see bad machines where there is a clear pattern of crashes on a particular core, so this is probably worthwhile. But the most insidious cases are true silent corruptions where the detection is far removed in time and space from the cause.

Our best hope then is to apply screening programs once the machine is ejected to try to pinpoint the bad core. It's unclear as yet how effective this can be.

IV. Modeling a Single Faulty Chip

Trying to analyze the factors involved and their mathematical inter-relationships might help us be more focused in our approach. This section contains a rough first cut; much remains to be developed/improved, but it does seem to provide some useful insights already.

Characterizing the properties of a single chip:

- R: rate of faults on a faulty chip
- K: number of faults required for detection (and removal) of a bad chip
- E: fraction of faults R that cause actual undetected user-visible Spanner errors (on reads or writes), as opposed to crashes or irrelevant faults (such as corrupting a cache line that gets flushed before being used again).

- D: Fraction of faults R that generate detectable errors (crashes, checksum errors, etc). Note that D and E are independent: a fault may cause a corruption and/or be detected, and some faults will be neither corrupting nor detected.
- H: Fraction of CPU time that any error signal-producing program (including Spanner) is running.
- Q: Fraction of CPU time that Spanner is running.

Then for any bad chip i:

- $d_i = R \cdot H \cdot D$ is the rate of detectable errors.
- $k_i = K/d_i$ is the time required to detect and remove it.
- $e_i = R \cdot Q \cdot E$ is the rate of Spanner errors.
- $n_i = k_i \cdot e_i$ is the number of silent corruptions in Spanner (writing or serving incorrect data).

That is,

$$\begin{aligned} n_i &= K/R \cdot H \cdot D \cdot R \cdot Q \cdot E \\ &= K \cdot E \cdot Q/D \cdot H \\ &= K \cdot E/D \cdot H/Q \end{aligned}$$

Intuitively, the number of corruptions served by a bad chip is:

- the number of corruptions required for detection multiplied by
- the ratio between the fraction of faults that cause corruption and the fraction of faults that are detected multiplied by
- the ratio between the fraction of time corruption-detecting applications are running and the fraction of time Spanner is running.

Some observations based on this formulation:

1) The number n_i of Spanner corruptions served is independent of the error rate R, which is somewhat counterintuitive.
2) Increasing D (faster detection): The more we can prevent errors and turn them into detection events (e.g. a checksum failure that prevents a corruption and registers a data corruption event), the fewer corruptions we serve. We tend to think of the role of such checks as preventing the errors themselves, and insufficiently value their role in preventing future corrupted results.
3) Increasing H (more detectors): The more we can leverage other services as "detectors", the fewer corruptions we will serve. The more we can put things like checksum checks in hot code paths, the better. Also, how should we trade off running screener/detector processes (that have no other utility) versus their cost in resources?
4) Reducing K (faster ejection): Obviously, the fewer the number of detected faults we need in order to diagnose a bad chip, the fewer the number of corruptions we serve.
5) What about Q? Given constant total workload, we could reduce Q on one machine but would have to increase it elsewhere to compensate. For the purposes of this single-chip analysis, we view Q as constant.

What accounts for the unintuitive result that corruptions are independent of the fault rate on a bad chip? We only considered removal of chips because they were bad. In fact we also remove chips over time due to obsolescence. If we add the parameter

- Y: the time until a chip is removed from the fleet, regardless of faultiness.

Then

- $k_{i2} = \min(k_i, Y)$ is the time until a chip is removed (whether bad or not), and
- $n_{i2} = \min(n_i, Y \cdot R \cdot Q \cdot E)$ is the number of corruptions served.

The question then is: how frequently do faults occur once a chip has gone bad? If a bad chip starts generating faults so fast that we eject it long before the lifetime of the chip expires, then reducing the fault rate on bad chips by a factor of 10 may not reduce the number of corruptions served.

Unfortunately this seems to be precisely the regime in which we are operating, at least for some of the cases we are detecting: regardless of whether a newly installed chip is bad or a chip goes bad over time, it takes us some number of weeks or months to eject it. Meanwhile, chips live in the fleet for some number of years.

V. The Future: Hardware versus Software Bugs

A common question is whether SDC represents a truly new or bigger threat, since software bugs are by no means uncommon. The answer depends on the type of application:

1) For software that has a small deployment, the low probability of SDC makes it unlikely to be a concern.
2) For software that has a significant number of software bugs, SDC will remain below the noise floor, even at scale.
3) For software that is highly reliable and deployed at scale, SDC is a significant problem.

For Spanner, it is already the case that a significant number of unique failures investigated by our crash triage team are caused by bad machines.

As we scale the deployment, the number of SDC events will go up, and the "bug antagonism" problem will continue to get worse: more time and resources will be spent on bad machines, leaving less time and resources for finding and fixing software bugs.

Software bugs tend to scale with the complexity of the software system and the size of its team. Software bugs tend to have a locus of failure, making them (relatively) easy to group and requiring a single fix. They scale sublinearly with the size of the fleet.

Hardware bugs scale with the size of the fleet and usually manifest in completely unrelated ways. Preventive measures can help limit the damage, and are improving with time and experience. But repair of corruptions is very expensive in engineering time. A steady stream of faults is death by a thousand cuts.

So which is the bigger threat, software bugs or hardware SDC? At the present time, I still believe software bugs are the bigger danger, in particular bugs that are low enough

probability to reach production but frequent enough to cause significant damage. Software bugs in the operating system and compiler can manifest similarly to SDC in that they violate the correctness contract of the underlying system upon which we build the database system. Spanner has encountered these as well.

If hardware SDC continues to increase and/or hardware bugs generate repeating but nevertheless non-deterministically infrequent faults, we may need to make bigger and more fundamental changes in our hardware and software infrastructure. This could involve hardware solutions like redundant execution (like the old Tandem NonStop systems) or designing failure redundancy into our software in more fundamental ways.

VI. Scaling Effects

When a system grows at a substantial rate, the per-CPU-second error rate must be continually driven down just to keep the rate of total errors constant. If errors scale with system size, a growing system is eventually overwhelmed by the failures. For instance, a certain fraction of SDC events consume large amounts of engineering time to mitigate or repair. If the absolute number of such events continues to grow, eventually engineering resources are overwhelmed.

This applies in both hardware and software. Spanner continues to grow quickly, due to organic growth of existing services, creation of new services, and migration of existing services onto Spanner. While we have made significant improvements in per-CPU-second SDC events, these improvements have been offset by service growth.

Similarly, as vendors scale up transistors per chip and scale down feature sizes, improvements in per-transistor fault rates may be overwhelmed by transistor growth.

In combination, these trends represent a major threat to Spanner and other mission-critical systems.

VII. Conclusions

SDC is a significant threat to the reliability of large-scale distributed systems, in ways that are both obvious and subtle. We have found the following software approaches to be effective ways of detecting and preventing SDC :

- Use end-to-end checksums to protect data when the semantics of operations allow them to be calculated easily and reliably (e.g. for filesystem reads or database mutations).
- Use fail-stop invariant checks when performing operations with complex semantics on data (e.g. non-trivial SQL expressions).
- Don't just protect big data structures. Also focus effort on components that have the largest "computational mass" – volume of data times computation on that data – since SDC tends to strike the CPU.
- Where possible, drive down the noise floor by eliminating bugs in other components.

For operations:

- Add centralized monitoring of (likely) SDC-implicating events, and include the likely machine source where possible to aid in ejection decisions.
- Co-locate high-reliability services so that the higher-fidelity screening they provide leads to faster ejection of faulty machines.

For hardware vendors,

- Minimize the number of bad chips, not the rate of faults once a chip goes bad.
- Invest in software detection methods, and share them proactively.

References

[1] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford. Spanner: Becoming a SQL system. In *Proceedings of the ACM International Conference on Management of Data*, pages 331–343, 2017.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 251–264, 2012.

[3] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar. Silent data corruptions at scale. *CoRR*, abs/2102.11245, 2021.

[4] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat. Cores that don't count. In S. Angel, B. Kasikci, and E. Kohler, editors, *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, pages 9–16. ACM, 2021.

[5] R. Pang, R. Cáceres, et al. Zanzibar: Google's consistent, global authorization system. In *2019 USENIX Annual Technical Conference*, pages 33–46, July 2019.