Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Chaos Engineering for Databases

Author: Thanh Long Tran (2535163)

1st supervisor: Hannes Mühleisen
daily supervisor: Mark Raasveldt (CWI)
2nd reader: Peter Boncz

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 12, 2020

"I am the master of my fate, I am the captain of my soul"

from Invictus, *by William Ernest Henley*

Abstract

No hardware is immune to random faults. Numerous studies have shown that faults in main memory happen quite frequently either due to external causes or internal damage. Industrial hardware are equipped with error correcting codes but even those do not provide full protection. Databases rely on the main memory to perform calculations and present accurate results. It is not well understood, how a database system can handle bit-flips in memory. To get a better understanding of these faults and their effects on databases we adapt chaos engineering to them, a fault tolerance evaluation methodology defined by Netflix. We run large scale experiments on databases that involve injection of bit-flips into running queries. We find that silent data corruption or even database file corruption has an unexpectedly high probability of happening. With the increasing trend of devices coming with more and more memories and the world increasingly depending on data, this issue will only get worse in the future. A prototype named AHEAD tries to remedy this by attempting to detect bit-flips during query execution. It does so by hardening data using AN encoding. We evaluated this prototype with our experiments and found that AHEAD has great potential to prevent silent data corruptions. However, AHEAD has several flaws that need ironing out before this solution can be considered mature. In this thesis we present our approach to detecting bit-flips during query runtime. We implemented two different techniques into an existing database called DuckDB. After evaluating the protected DuckDB with our chaos experiments, we find that one of the two techniques can prevent up to 96% of all silent data corruptions. However, the current implementation has a large performance overhead. With enough performance improvements we believe that bit-flip detection would be a valuable addition to databases designed for analytical workflows. The results of our chaos engineering experiments clearly show that the prevention of silent data corruption is possible.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1 Introduction

1.1 Context

Hardware and software failures can happen anytime and without any notice causing degradation or outages in entire complex systems. For most services, high availability is one of the most important aspects of product quality. An unexpected hardware fault could cause system failures resulting in costly damages. Various measures can be taken to make systems resilient to failures, but without testing them in real world scenarios we cannot be confident in the fault tolerance of such measures.

With the rise of large-scale distributed systems, it becomes increasingly more difficult to ensure fault tolerance, as the number of ways these systems can fail also increases (2). To better understand how their entire system behaves in a chaotic environment, Netflix have created a tool called Chaos Monkey(9) to subject their production system to real-life fault scenarios. Subsequently, they have developed their own method for fault tolerance testing called Chaos Engineering(5).

We observe an increasing trend of the world depending on data. We rely on database systems to consistently store an manage large amounts of information for us. There have been many studies in production environments pointing out the relevance of hardware memory faults(17, 25, 26, 31, 32, 33, 38, 40, 43). However, the nature of hardware memory faults and their effects on databases are not well understood. More often than not, databases are not developed with the possible chaotic environments in mind. Commodity memory hardware do not usually come equipped with error correcting codes (ECC) and even if they do, they have their limitations and are not able to protect against all of the faults. A bit flip in the main memory can cause several different issues. The most severe problem these faults can possibly cause is silent data corruption.

1.2 Research Question

We defined the following research questions for our thesis work.

- How can we adapt the disciple of Chaos Engineering to databases to evaluate fault resiliency of databases?

In order to better understand how they behave in a chaotic environment, we need to perform chaos engineering experiments on database systems. Chaos engineering is a well defined methodology making it suitable for adapting to other areas in the field.

- How susceptible are modern database systems to hardware memory faults and silent data corruption?

By adapting chaos engineering to databases, we are able perform chaotic experiments on them. With these experiments we aim gain insight into how databases behave under chaotic environments.

- What measures can we take to make databases resilient to these faults and what are their costs?

Once we have a better understanding of the nature of hardware memory faults, we need to consider the possibility of mitigating the effects. These defensive measures will come with performance and storage costs that need to be evaluated.

1.3 Research Method

To adapt chaos engineering to databases we need to simulate a chaotic environment for databases to run in. For this we inject bit flips into the main memory and use TPC-H to reproduce a production level workload. Emulation of bit flips is possible with fault injection tools. Although there are numerous existing fault injection tools, we decided to create our own from scratch to meet our needs. We designed an experiment that evaluates the fault resiliency of databases by running queries while injecting faults. We present the results of these experiments in this thesis. Additionally we present our implementation of bit flip detection integrated into DuckDB and evaluate its performance in preventing silent data corruption. We also evaluate the detection performance of AHEAD, a prototype database that also claims to be able to detect bit flips during query execution.

2

Background and Related Work

2.1 Terminology

The definitions of faults, errors and failures can be unclear but generally speaking a *failure* is caused by an *error* and an error is caused by a *fault*. For example, a malfunctioning database software returning incorrect results is a failure, caused by corrupted memory content which is an error. The root cause of such an error can be particle strike that changed the content of the memory.

In the context of memory errors, the number of affected bits can be different based on the type of the error. If the root cause affects more than one bit, it is called a *multi-bit* error. Otherwise it is called a *single-bit* error. Multi-bit errors are typically chip-level errors, meaning that they are caused by a faulty row, column or an entire bank. These are usually caused by hardware defects and permanent damage. They are called *hard* or *non-transient* errors and they are easier to detect because they are repeatable. *Soft* or *transient* errors however can be much harder to detect because they are only a temporary error usually caused by an external fault, such as magnetic interference or a particle strike.

High-grade and industrial memory hardware is often equipped with an error correcting code (ECC) that can detect and even correct memory errors. These codes cannot correct every error as they are limited to correcting faulty words that have a low Hamming distance to the original word. Hence memory errors can be categorized into *correctable* errors (CE) and *uncorrectable* errors (UE).

2.2 Memory Errors in the Field

There have been studies on soft memory errors under controlled environments(7, 19, 29, 34, 46, 47). However, the results of these studies are limited because they were not conducted on large scale production environment running real world applications. To better understand the nature of these errors in a real world environment, prior work has done surveys on large data centers(26, 27, 32, 38) and supercomputers(17, 40, 43).

A three months long study on more than 200 Ask.com machines with a total of 800 gigabytes of memory detected 8288 suspected hard errors and 2 suspected soft errors(25). Later the study on the same machines, but over nine months recorded an error rate of 2006 FIT without error correcting codes (ECC) and 1000 FIT with ECC. They also studied the effect of errors on applications and found that 48% of the errors were activated, meaning the they were accessed in memory. And 62% of the activated errors caused the application to crash or give an incorrect result(26).

Another study on the Google server fleet was done over 2.5 years on 6 different hardware platforms. They observed FIT rates of 25,000 to 70,000 per megabit, which is much higher than assumed(38). One factor that could contribute to the high error rate is the usage of older technologies such as DDR1 and DDR2. They also found that if a device had an correctable error, the increase in the probability of having an error in the same month can be more than 90x and the increase in the probability of having one the next month can reach 200x. They also come to the conclusion that error rates are likely to be dominated by hard errors rather than soft errors.

(32) studied the entire Facebook server fleet over 14 months examining 6 different platforms. They do not report an FIT rate for the devices, but found that overall 9.62% of the servers were affected by correctable errors. Each month 0.03% of all the errors are uncorrectable and there is a mean average of 497 correctable errors per month. Of all the errors 7.8% are categorized as *spurious* errors. These errors do not share a common component and do not repeat making them highly likely to be transient errors. Spurious failures affected 56.03% of the servers with errors. They justify the lower error rate compared to the study on Google servers with newer technologies and a more aggressive repair policy. They performed analysis on page offlining and found that it could reduce the error rate by 67%(32).

There are some work on studying memory errors in super computers. A study that involved a Blue Gene supercomputer at Lawrence Livermore National Laboratory, a Blue Gene supercomputer at Argonne National Laboratory and a high-performance computing cluster at the SciNet High Performance Computing Consortium found that 2.5-5.5% of nodes were affected by errors(17). The error patterns in the study link the majority of memory banks to hard errors and 15% of the errors activated Chipkill, a multi-bit ECC(14). Schroeder et al. also found that an incident of an error greatly increases the probability of a multi-bit error. They have also tested page retirement and found that up to 96% of all errors could be avoided. A study on Cielo, a supercomputer at Los Alamos National Laboratory, calculated a error rate of 0.044 FIT/megabit, which translates to about one error every 11 hours across the whole system(40, 43). Their data shows that about 50% of all faults are transient.

There were also studies on memory errors in commodity systems. Experiments on the susceptibility of commodity systems have shown a possible activated error rate of 687 FIT(31). Another study analysed data from the Windows Error Reporting (WER) system. WER reports contain information about hardware faults that lead to system crashes. They found that the probability of a first failure due to a bit flip in kernel memory in 30 days is 1 in 1700(33). This probability increases by more than two orders of magnitude after the first failure was observed. Unlike industrial hardware, commodity systems do not typically come equipped with ECC making the more susceptible to failures caused by bit flips.

From these studies we can conclude that the probability of a hardware fault in memory is higher then we would expect. Although hard errors dominate, which is not surprising, transient errors still do happen quite often. These transient errors are harder to detect and can cause damaging data corruption, therefore the importance of protecting against these faults is not negligible.

2.3 Chaos Engineering

Netflix is one of the biggest streaming platform with a large infrastructure of distributed systems. To ensure high availability of their system, they have created a set of tools called the Netflix Simian Army(9). Chaos Monkey, the first member of the army, was a tool that randomly selects virtual machine instances in production and shuts them down to see how the whole system would respond to such a failure. Later they created the Chaos Automation Platform (ChAP), which is a complex system that safely automates chaos experiments in their production environment as well as generating experiment designs for new components(10)(6). This eventually led to the definition of *chaos engineering*, the disciple of experimenting on a software system in production to test and build confidence in its resiliency(5). Basiri et al. define four principles that embody chaos engineering.

Firstly, **build a hypothesis around steady-state behaviour**. Engineers at Netflix need to know what it means for their system to "work properly". They use the number of streams started per second as a metric to indicate the system's overall health. Any observed irregularity can be an indication of a degraded system performance. In case of database systems we would want it to successfully answer a query and give the correct result every time.

The second principle is **varying real-world events**. Any event in the real world that can happen, will happen and is a good candidate to create experiments for. Netflix engineers used inputs such as virtual-machine termination, request latency between services, request failure between services or making an entire Amazon region unavailable. In this thesis we focus on transient hardware faults in the main memory that can cause silent data corruptions.

The third principle is **running experiment in production**. The Netflix infrastructure is too big making it impossible to deploy the whole system onto a single machine. Traditional software-testing approaches are insufficient to fully recreate the production environment. They do not believe in reproducing the system in a test context as there will always be slight differences. For databases we will use TPC-H[1] database benchmarking tool to reproduce production workload.

Lastly, chaos experiments are **automated to run continuously**. The distributed system at Netflix is constantly changing as the engineers continuously update different services at the same time. To ensure confidence in the results of these experiments, they need to be run repeatedly as the system evolves. It is near impossible to automate experiments for databases and transient errors as the memory usage pattern changes all the time and also depends on the operating system. Statistical experiments similarly to the ones done in this thesis could be run in a weekly or monthly fashion.

2.4 Data Resiliency, Error Detection and Correction

Techniques to protect memory hardware from errors already exist. Memories in industrial devices often come with some kind of error detection and correction code (ECC) to protect against hardware errors. For example single error correct double error detect (SECDED) codes can correct single-bit errors and detect multi-bit errors. A stronger protection called chip-kill can correct multi-bit errors and is also able to handle completely broken memory

[1]http://www.tpc.org/tpch/

chips(14). Recent work has introduced Multi-ECC, a multi-bit error correction code that has a lower overhead and lower power consumption than chip-kill(18).

Errors that can be corrected by these codes are called correctable errors and errors that may be detected but not corrected are called uncorrectable errors. Correctable errors are mostly invisible to applications but comes with a small overhead of logging and the performed correction. A read access to an uncorrectable error can often lead to a catastrophic failure. Most commodity devices like desktops and laptops do not come with ECC. An error that normally be correctable will act as uncorrectable errors in those machines and can cause a system failure as we have seen in (33).

To our knowledge transient DRAM errors did not receive much attention in the context of database research. Existing work relies on resilient frameworks, protect against detectable errors or only address smaller aspects in data management(11). One recent work introduced a lightweight resiliency-aware data compression that utilises AN-encoding(22, 23). Later they created AHEAD, a prototype columnar store, that uses this compression to protect the data from bit flips during query execution(24). It claims to be able to detect multi-bit flips while having a lower overhead compared to dual modular redundancy. Although calculations of the probabilities of silent data corruptions show promising results, the prototype has not been tested in a chaotic environment with bit flip injection.

2.5 Fault Injection

Although the concept of chaos engineering is recent, the practice of fault injection to test the resiliency of a system is not new. Fault injection tools can fall into two main categories: hardware implemented fault injection (HWIFI) and software implemented fault injection (SWIFI)(30).

HWIFI tools require additional hardware that is usually specialized for injecting faults into other hardware. This can be done without contact using radiation or with direct contact to the circuit pins. RIFLE(28) and MESSILANE(3) are both direct contact HWIFI tools and FIST (Fault Injection System for Study of Transient Fault Effect) employs both contact and contactless methods to inject faults(21). FOCUS is an automation environment designed for tracing the effects of a transient fault(13).

Nowadays, SWIFI tools are more popular because they are easier to use, create a more controlled environment and they do not require expensive hardware. Software fault injection can happen during compile-time and during runtime(30). During runtime an injection can be triggered in three different ways:

- Time-out: The injection is triggered by a timer.
- Exception/trap: A hardware exception or software trap transfers the control to the fault injector.
- Code insertion: Instructions are added to the target program that allows fault injection.

There are a number of fault injection that already exists. FERRARI (Fault and Error Automatic Real-Time Injection) is a tool developed at the University of Texas at Austin, that uses software traps to inject CPU, memory and bus fault(20). FTAPE (Fault Tolerance and Performance Evaluator), developed at the University of Illinois, is a tool that can inject faults into user-accessible registers in CPU modules, memory locations and the disk subsystem(37). FIAT (Fault Injection-based Automated Testing), developed at Carnegie Mellon University, is an automated fault injection environment, that gives the experimenter the ability to design fault experiments(4). Xception, developed at University of Coimbra, is a tool that uses the advanced debugging and performance monitoring features of modern processors to inject fault(12). DOCTOR is an integrated software fault injection environment developed at University of Michigan, that can inject CPU, memory and network communication faults(39). EXFI is a fault injection system for embedded microprocessor-based boards developed at Politechnico di Torino, Italy(8). NFTAPE, developed at the University of Illinois, aims to provide multiple fault injection tools for distributed systems(44). GOOFI (Generic Object-Oriented Fault injection), developed at Chalmers University of Technology in Sweden, aims to provide a user-friendly fault injection environemnt that supports multiple injection techniques and multiple systems(1). Its successor, GOOFI-2, includes a large number of improvements and extensions(42).

There are three additional categories fault injection techniques can fall into. Simulation-based fault injection involves simulation models developed with a hardware description language such as Very High Speed Integrated Circuit Hardware Description Language (VHDL). In emulation-based fault injections, the circuit to analyze is implemented onto an FPGA. Then a host machine connects to the board and used to define fault injections. Finally, hybrid fault injection combines two or more of the previous fault injection types(45).

3

Fault Injection

To perform our experiments, we need a tool to inject bit flips into the memory of a running process. This chapter describes our rationale behind choosing to implement our own tool instead of using an existing one. We also describe the details of the implementation of the fault injection tool.

3.1 Requirements

The fault injection tool needs to be able to inject hardware memory faults into a running process. Different database systems use different amounts of memory and take a different amount of time to respond to a query. If we inject the same amount of bit flips to two different systems, it would not be a fair comparison for the database that uses less memory. To normalize the number of injected faults over space (amount of used memory) and time (runtime of a query), we define the metric called *fault rate* with the unit of faults/megabytes/second. This input parameter ensures that the amount of injected faults increases in proportion to the used memory and time taken to answer a query.

The faults must be continuously injected into the process over time and not just once. Meaning the faults must be injected periodically. The period can be calculated with the following equation, where P is the time between each fault, M is the current size of the used memory and r is the fault rate provided by the user.

$$P = \frac{1}{M * r}$$

Additionally the tool must make sure that at least one fault is injected into the target process. This is because if the input fault rate low enough, the time between faults might be longer than the time it takes for the query to finish.

Finally, the tool should record all outputs needed to evaluate the experiments. To be able to tell if there was a silent data corruption, the standard output of the target process needs to be captured. Furthermore, the exit code of the process, the termination signal that caused the process to stop and the standard error output all need to be captured to give us insight into the reason of a failure of a query.

3.2 Technique

As we have seen in Section 2, there are many ways to inject bit flips into the main memory. Hardware-based fault injection requires specialized hardware and it is almost impossible to create a fully controlled fault injection environment. We need to be able to fully control the target address of the injected bit flip, so hardware-based fault injection is not suitable for our experiments. A virtual machine based fault injection tool would let us have full control over memory of the guest operation system. However, we want to run queries millions of times, which can naturally take a long time and the overhead from virtual machine based fault injection can significantly increase the runtime of the experiments. Therefore software-based fault injection is the most suitable for our experiments as it can give us the most control with a low overhead. However, most existing software implemented fault injection (SWIFI) tools require modifications to the kernel or root permissions on the machine or they are specialized for a specific hardware or operating system.

To run experiments on many different database systems, we have decided to implement our own generic fault injector that can work on any process on Linux systems. Our tool does not need any special permissions allowing us to run the experiments on a cluster in a safe manner isolated from the other processes. We want to flip bits in the area of the memory that is used by the database system. We utilize the `ptrace(2)` Linux system interface as it lets us access and modify the internal state of another process(41). The interface was originally designed for debugging other processes. The data flow between the debugging and tested processes is checked by the kernel so tampering with the process cannot affect other processes making our experiments self contained.

The `ptrace(2)` interface can be found in most Unix systems making the injection tool near operating system independent for Unix based systems. One only needs to be able to compile the source on the target system. Windows systems have similar system calls that read from and write to the memory of another process. In theory, the injection tool should be relatively easy to be ported to be Windows compatible.

3.3 Implementation

To utilize `ptrace(2)` and other system interfaces and minimize performance impact on the tested process, the tool is implemented in `C++`. The source code of the injector is publically available online[1]. We used the `C++` 11 standard for the injector. Any configuration described in this chapter can be set using command line options when running the fault injector. A high level overview of the fault injector can be seen on Figure 3.1.

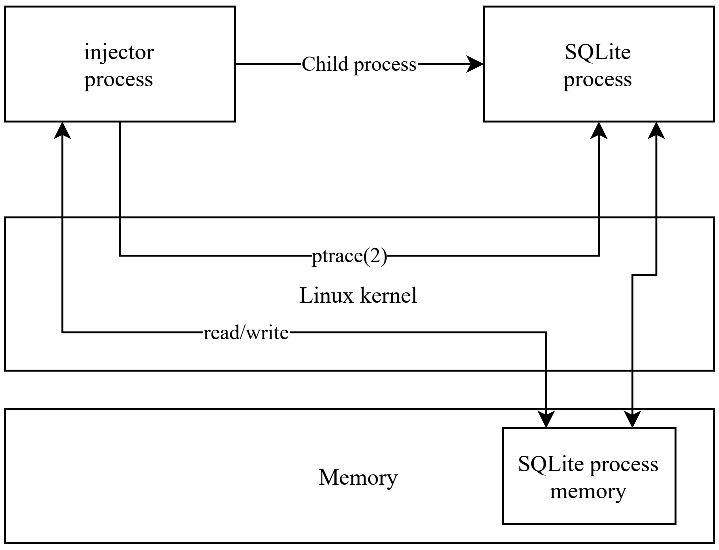


Figure 3.1: High level fault injector diagram

To debug a process it either needs to call ptrace with PTRACE_TRACEME to indicate that it is to be traced or the debugging process has to attach to the tested process with PTRACE_ATTACH. The former would require modifications to the database system's sources, while the latter does not. However, the kernel prevents attachment to another process unless the user has root permission or the tested process is the child process of the attaching process. We will run experiments in an environment without root privileges, so the injection tool runs the database system process as a child process in order to be able to write to the memory of it. This way, it will not be able to access other sensitive parts of the memory that might crash the entire machine. The command to run a database query or server is passed to the injector through command line arguments and the child process is created with the `fork(2)` system interface.

Linux systems use virtual memory to make the management of the physical memory easier for the user. The injection tool needs to know which virtual memory address spaces

[1]https://github.com/Longi94/chaos_db

the child process uses to write bit -flips into the content of the process' memory. The maps file under the `/proc/<pid>` directory contains all the memory regions used by the process. The file describes the exact regions used for the heap, stack or anonymous regions used by a process. To parse this file, we use the existing proc_maps_parser[1] tool. The tool had to be slightly modified as in rare occasions the parsing process ends up in an infinite loop due to race conditions on the maps file.

Usually the heap contains the data that can be corrupted silently. However some binaries might use the `mmap(2)` system call to allocate memory, which usually creates anonymous mappings in the memory of the process. Even the traditional `malloc(3)` system call's implementation might use mmap() silently if the requested allocation size is large enough. The injector can only inject bit flips into the heap and the stack in its current implementation. We found that the size of the region of the memory used by the stack is constant and the actual stack only uses a small part of it. Therefore, we only inject faults into the heap in our experiments.

To inject faults into the child process, the injector runs on its own clock and at each tick it decides whether to inject a fault or not based on the input fault rate and the size of the memory used by the child process. When calculating the size of the used memory, only regions where the tool is instructed to inject faults into are accounted for. By default, faults are injected periodically at a consistent interval that is calculated using the formula in the previous section.

If the provided flip rate is low enough, a database query might finish before the injector had a chance of injecting a bit flip. To ensure injecting at least one fault the minimum runtime of a query has to be known by measuring it without fault injection prior to the experiments. When the process starts, at each tick there is a small probability that the first fault is injected. This probably increases with time and reaches 1 before the query process finishes. This has the added benefit of randomizing the start of the periodic fault injection, making it more realistic.

Faults do not occur in a fixed rate in real life. Therefore, the tool offers the option to randomize the interval between each bit flip to simulate a more realistic environment. However, using this option in our experiments would result in each run being slightly different and gives us less control overall over the experiment input. We find that enabling this option is not useful for case, so we leave it disabled.

The process of injecting bit flips is fairly simple. The sequence diagram of the process is shown on Figure 3.2. First the injector attaches to the child process using `ptrace(2)` with

[1]https://github.com/ouadev/proc_maps_parser

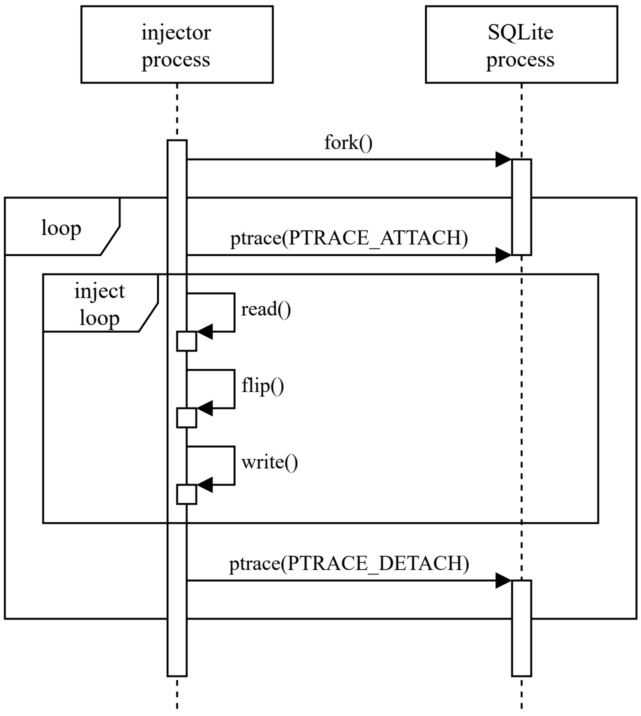
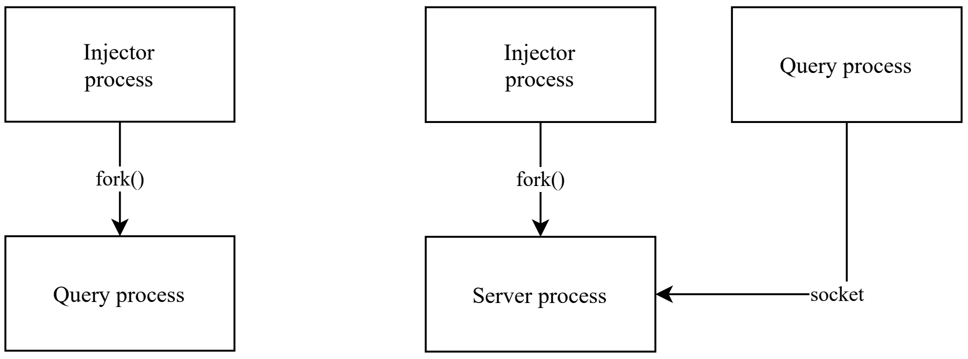


Figure 3.2: Process of injecting a fault

PTRACE_ATTACH. This causes the child process' execution to stop. The injector opens the `/proc/<pid>/mem` file as it can be used to access the pages of a process' memory. It chooses a random address based on the content of the maps file. Then it reads a single byte at the chosen address using `read(2)`, flips a bit inside the byte with a XOR operation and writes it back to the same address using `write(2)`. Finally, the process detaches from the child using `ptrace(2)` with PTRACE_DETACH. To avoid slowing down the experiment with frequent ptrace calls, the injector uses its own timer to group multiple injections into a single ptrace call, hence the nested loop structure in the diagram.

Once the query has finished running, the injection tool records the information about the process including the exit code, the termination signal, the number of faults injected, whether the process timed out or not and the maximum recorded size of the memory. We also need to save the standard output of the query so we can compare it to the expected output. During our experiments we found that saving the output to a file produces millions of files and can cause a file system degradation. This impacted the performance of the queries and could only be solved by reinstalling the device. To avoid this issue the all the result and output is saved into a SQLite database.

We differentiate two ways a database query can run that affects how our SWIFI tool



(a) Embedded system **(b)** Server-based system

Figure 3.3: Difference between injecting faults into embedded and server-based database systems

operates. This difference is demonstrated by a high level diagram shown in Figure 3.3. Most of the time, embedded databases, such as SQLite3, have a single process that executes the query and also delivers the result. The tool will simply run this query as a child process and inject bit flips into its memory (shown on Figure 3.3a). Other database systems are server-based, therefore queries will involve two or more processes. The client process initiates the request and presents the results, while the server process does all the heavy calculations. In this case the fault injection tool runs the database process as the child process, while the query itself is initiated by a separate process (shown on Figure 3.3b). The injector opens a TCP socket where it will wait to be notified when the query is started. It will then start injecting bit flips and wait for another message that tells it to stop.

3.4 Limitations

The injector uses the virtual address space of the memory to inject fault. Without knowing how the kernel translates these addresses to physical addresses it is impossible to emulate some other faults like row and column faults. It might not even be possible to reliably convert a virtual address to a physical address in most operating systems.

Injecting faults into bigger database system like PostgreSQL might involve multiple processes as they can spawn their own child processes creating a complex tree of processes. While not impossible, it becomes troublesome to discover these processes and keep track of the shared memory spaces. The overhead caused by this might even make it infeasible to run large scale experiments.

With this method it is currently only possible to inject transient faults into memory. Injecting permanent faults could possible in single step mode. However, this mode is very slow as the CPU is debugged step by step. This would drastically increase the runtime of a database query making it unsuitable for large scale experiments.

As mentioned before, the injector can only inject faults into the heap and the stack in its current implementation. This means that systems that use memory maps and create anonymous areas avoid the bit flips. Additionally, the injector needs more information about the stack if we wish to inject faults into it.

4

Experimental Setup

This chapter describes how the experiments are carried out and the environment they are carried out in. The aim of these experiments is to see how database systems can respond to unexpected bit flips in memory. In particular, we want to show that silent data corruptions are possible.

4.1 Environment

We needed to execute rather long running database queries millions of times. All experiments are ran on **rocks2** machines on the SciLens cluster[1] to minimize the runtime allowing us to run millions of database queries in a relatively short amount of time. These machines are equipped with one Intel® Xeon® E3-1270 v3 processor and 64 GB of main memory. Experiments were ran under 64-bit Fedora 30 and all the binaries needed for the experiments were compiled on the cluster using GCC 9.1.1.

4.2 Setup

A list of database systems used in the experiments is shown in Table 4.1. SQLite is the most used database engine in the world(15), making it statistically the most vulnerable to bit flips in the real world. DuckDB is an embedded database similar to SQLite(36). However, DuckDB is designed for analytical workloads, making it much faster than SQLite but uses more memory to perform queries. We also implemented simple mitigation techniques into DuckDB to try and detect bit flips in memory during query execution. Since AHEAD was

[1]https://projects.cwi.nl/scilens/

profiled without the injection of memory faults(24) we ran experiments on it to see how well the hardened data fares against bit flips.

Name	**Version**
SQLite	3.29.0
DuckDB	638c98d6e1efd2f5cabecfc61791fb1505c76edd
AHEAD	ca98604f339276080991171fec36607cb4f59c77

Table 4.1: Databases used in the experiments

Since DuckDB was in early stages in development and AHEAD is an unreleased prototype, the version used is identified by the commit hash created by the git version control system. We needed to slightly modify the AHEAD source to successfully compile it for the operating system on the SciLens cluster. The version of AHEAD is the latest commit on the master branch of the original repository at the time of running the experiments. The source code of the modified AHEAD system is available online[1]. To give bit flips a higher chance of affecting the process we need queries that run long enough to give time to inject faults. To achieve this, we used the TPC-H benchmarking tool to generate a large enough workload.

4.3 Sample Evaluation

There are many things that can happen when a database process encounters a bit flip in its memory. We categorize the outcomes into these categories: *ok*, *abnormal*, *crash*, *incorrect*, *corrupted* and *timeout.*

There is a chance that a process goes unaffected by bit flips as these transient errors can simply be overwritten by the process or not accessed at all. In these cases, the faults are completely undetected and the query will return the expected result. These results are classified as *ok.*

We are most interested in *silent data corruptions* (SDC), as they are not detectable if the expected output is unknown. SDCs can result in inaccurate query results, which can cause misbehaviour in critical systems. For example, SQLite is deployed with aircraft software(16), where minor changes in accuracy can be catastrophic. Since in the context of the experiments the expected query output is known, we can detect SDCs and classify the result as *incorrect.*

[1]https://github.com/Longi94/AHEAD

The rest of the categories are all detectable errors, so they are less fatal as the system can attempt recovery when these events are encountered. Crashes caused by unhandled UNIX signals are categorized as *crash*. When the query process does not crash but returns with a non-zero exit code along with a usually meaningful message printed to standard error, we categorize the result as *abnormal*. As we will see in the experiment results, in very rare cases, a bit flip can cause a query execution to enter and infinite loop. For this reason, there is a limit on how long the query can run. If this time limit is reached, the process is terminated by the injector and the result is categorized as *timeout*.

The *corrupted* outcome is only applicable to experiments with write queries. The database can get corrupted when corrupted data is written back to it and in a worst case scenario, it can corrupt the whole database on disk. This can render any data inaccessible or worse, lose all the data stored in the database.

4.4 Running on the cluster

To orchestrate the entire experiment, we wrote python scripts to bring everything together. These scripts are available online aswell in the same repository as the source of the injector resides in[1]. On a single machine there are multiple components involved in running the queries.

The **runner** is responsible for initializing the database, running the query with the injector and then cleaning up after running the query. If the tested database runs as a server it is also responsible for running the server process. Initializing the database involves copying a prepopulated database to a directory so the sample can be fully separated from other samples.

The injector outputs information to standard output that needs to be collected. To avoid filling up the injectors buffer causing it to hang, the **monitor** runs on a separate thread to constantly read and store information from the standard output of the injector. Once the query is done, the monitor will evaluate any information collected from the sample to classify the result to a category as mentioned in section 4.3.

The **orchestrator** is responsible for coordinating the whole experiment on a single machine. It uses the runner to run queries thousands of times and store the results returned by the monitor in a SQLite database. It will run multiple samples at the same time using python's multiprocessing module. Each sample will run in its own directory to fully separate them from each other. When a server-based database system is tested, the injector

[1]https://github.com/Longi94/chaos_db

opens a TCP socket. To avoid port collision the orchestrator creates a pool of port numbers for both the injector and the database server. This way multiple instances of the database server can run at the same time. However, sometimes the operating system might keep ports reserved for a while after terminating the server process. While the injector handles this, the database server might fail to initialize and keep running without the injector knowing. The runner will try to connect to the server multiple times and if the limit is reached, the orchestrator will simply skip the sample. A more complex handling of this issue could be implemented on the future.

Finally, to leverage the power of the cluster we wrote a simple python script to run the orchestrator on multiple machines. The script uses clush[1] to run commands on multiple machines from the master node. At the end of an experiment, it copies all the SQLite databases from the machines to the master node using clush in file copy mode and combines them into a single database file.

4.5 Input Variables / Workload

To simulate real world workload in the database system, we use the TPC-H benchmarking tool as it simulates an ad-hoc querying environment(35). We use a scale factor of 1 to generate the database tables, that are later imported into the database. We use the TPC-H 1 query to run bit flip experiments on.

We want to get a good idea of the chances of silent data corruptions and other possible results and how the fault rate can affect this probability. First we run the experiment for a specific database with a fault rate of 10^{-1}. For each consequent experiments we increment the exponent by 0.25 and the highest fault rate we test for is 10^{2}. For each fault rate we run the query 97500 times. This is because we would want to have around 100000 samples for each experiment, however, the experiments were run on 13 machines and 13 is not a divisor of 100000.

Although our work mainly focuses on read-only queries, we additionally run experiments on write queries to see whether the data in the system can be corrupted by faults or not. For this we use the update statements generated by the TPC-H benchmarking tool. For the updates we chose a scale factor of 50 to make it large enough to have the query run for long enough. For now we only run update experiments on SQLite. SQLite uses memory mapped files on disk, therefore update queries perform considerably large amounts disk I/O operations. This means that queries running in parallel on a single system would

[1]https://clustershell.readthedocs.io/en/latest/index.html

overload disc I/O and result in major performance impact. We are unable acquire as many samples as we could with read-only queries. We run the update query 13000 times for each query.

Since SQLite is a single-file database system, checking the file for integrity errors is fairly simple. After the update queries we can do a quick hash calculation on the database file. If the computed hash value is correct, we know that database is not corrupted and the updated records are correct. If the hash is not correct, than the sample falls into the *incorrect* result. Additionally, if the calculated hash is not correct, we do additional integrity check to see if the database file got corrupted or not.

5

Initial Results

In this chapter we present the results of our exploratory experiments and if possible, explain why we received these results. All experiments described in this chapter were performed on unmodified database systems in order to establish a baseline for the affects of memory faults. The experiment results and the generated diagrams in the figures are all available online in the form of SQLite databases and Jupyter notebooks[1].

5.1 SQLite

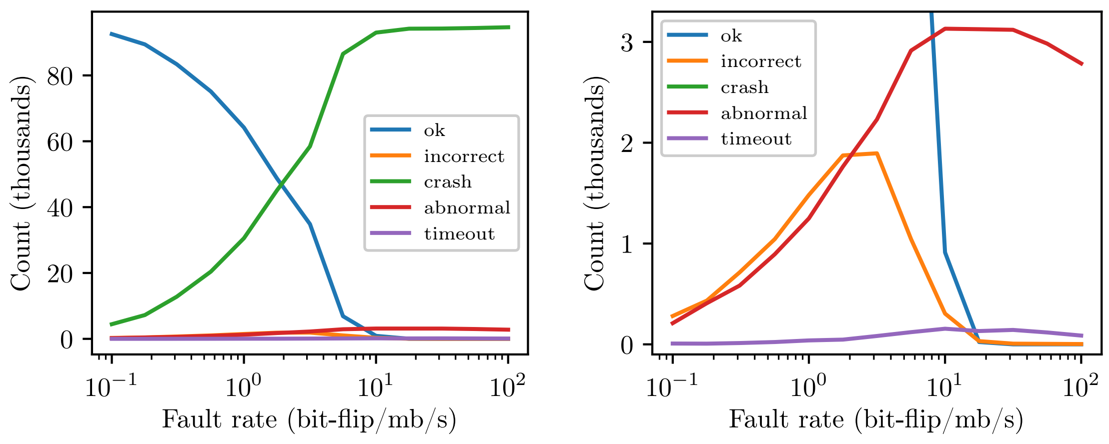
5.1.1 Read-only queries

Figure 5.1a shows the plot of the results of running TPC-H 1 queries on an SQLite database. Each line represents a type of result described in section 4.3. The X axis represents the bit flip rate in bit flip/megabyte/second unit and is in a logarithmic scale. The Y axis is a linear scale of the number of outcomes for the given result. There are 97500 samples for each fault rate. The rest of the fault rate plots are all set up similarly.

As we increase the rate of bit flips in memory, the crash rate of SQLite rapidly increases reaching 95.3% at the fault rate of 10. SQLite only uses around 2.5 megabytes of heap memory and uses memory mapped files for the actual data. Since little memory is used there is a high chance that a bit flip happens in an area critical for execution causing a high rate of crashes. Over 90% of the crashes are caused by a segmentation fault signal, which occurs when a bit of a pointer is flipped causing it to point to an inaccessible memory address.

Abnormal results are the next most common error results reaching 3.2% also at the fault rate of 10. The rate of abnormal results start to decrease slowly as crashes dominate at

[1]https://github.com/Longi94/chaos_jupyter



(a) All results

(b) Zoomed in

Figure 5.1: SQLite results

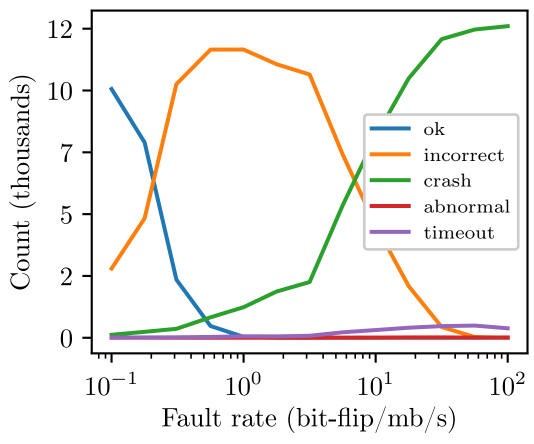
higher fault rates. SQLite does have some resiliency features as it can detect malformed disk images or disk I/O errors and abort execution when it does.

Figure 5.1b focuses on incorrect results, by scaling up the graph by the X axis. It is clear on the zoomed in plot that although silent data corruptions (incorrect results) are not common, there is a slight chance that they will happen due to a bit flip. The probability of SDCs peak at around 2% between the fault rate of 1.778 and 3.162. It quickly starts to decrease and tend to 0 when correct results also reach 0. Since SQLite uses memory mapped files on disk, the bit flips will not affect the persistent data. The computed result can only be affected by modifying the intermediate data in the main memory. It is likely that these intermediates are rather short lived, giving the injector less chance to hit it. This can be an explanation for the low rate of silent data corruptions as bit flips do not happen on disk.

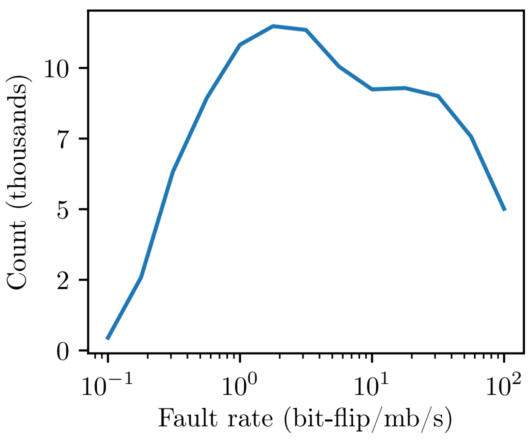
There is a very small probability of the process entering an infinite loop and in the case of SQLite the probability stays below 0.16% for all fault rates. Sometimes this results in the SQLite binary printing out the same row in the infinite loop which can rapidly fill up the I/O buffer. It is unclear what the conditions are for this to happen.

5.1.2 Write queries

The results for update queries are shown on Figure 5.2a. Although the curve of the lines resemble Figure 5.1a, the number of incorrect results is a lot higher than expected. The rate of incorrect results reaches almost 90% early on. There are a lot of different messages printed to the standard error output by the SQLite process, therefore we cannot draw a definite conclusion to why there are so many incorrect results. The logs indicate the most common error to be a malformed disk image or malformed database schema. Since SQLite uses little memory, it is possible that the database schema, that is loaded into the memory for update queries, takes up a relatively large part of the memory.



(a) All results



(b) SQLite database corruptions

Figure 5.2: SQLite results for update queries

The graph for the rate of database corruption is shown on Figure 5.2b. We observe that the chance of the SQLite database file getting corrupted reaches 88% at the fault rate of 1.778. At that rate, most of the queries received 8 fault injections. It is quite surprising how such a low amount of bit flips can almost guarantee a corrupted database. On higher fault rates this probability starts going down. This is likely due to the rapidly increasing rate of crashes. The process has less time to write anything to the file on higher fault rates.

Overall, we can conclude that even though SQLite uses little memory, it is still vulnerable to memory faults. Considering that it is the most widely used database in the world, the 2% chance of getting a silent data corruption translates to a relatively frequent rate. Write queries looked especially susceptible to silent data corruption or entire data loss due to database integrity corruption. This is quite concerning considering SQLite is the most widespread database in the world.

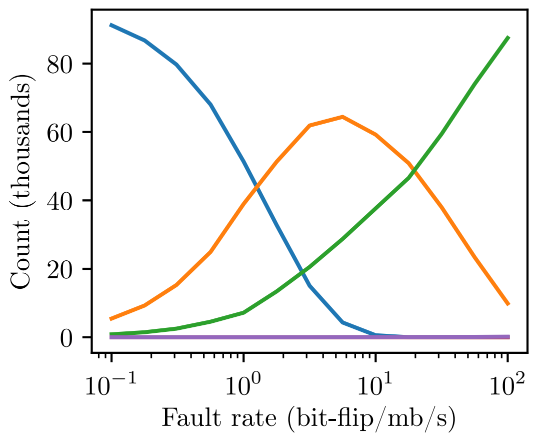
5.2 AHEAD

To our knowledge, AHEAD is the only recent work that focuses on resiliency against bit flips in databases. However, the prototype Kolditz et al. have built has not been actually experimented on with injected bit flips in memory. They have only run performance measurements and compared it to the unprotected version of the prototype. The reasoning behind this is that they have calculated the probability of a silent data corruption so fault injection experiments are not necessary(24). However, this probability might not reflect reality. We run our own experiments with our fault injector to see how the AHEAD prototype actually fairs in chaotic environments.

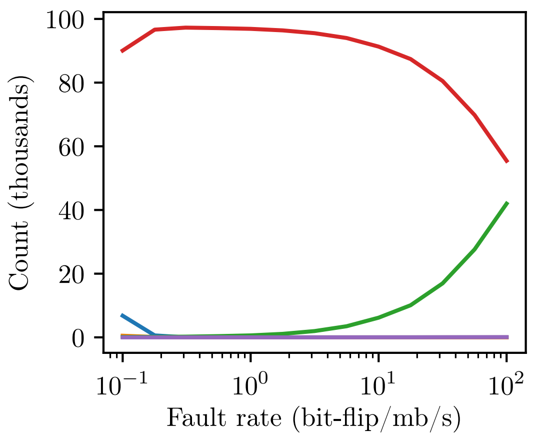
Data that is protected by the AN-encoding is called hardened data. Data is encoded on disk, which has the advantage of it being protected against bit flips at any point in time during query execution. But it also means that hardened data file format differs from non-hardened data, which can complicate the storage file format. AHEAD has five query processing variants. The *normal* variant is the baseline, data is not hardened and no bit flips will be detected. The *Early Onetime Detection* variant checks for bit flips the first time it touches the data. The *Late Onetime Detection* the detection may take place in a late stage of the query execution plan. The *Continuous Detection* variant has bit flip detection built into every single operator. There is an additional variation of the continuous detection where the operators also reencode the data with a new constant after checking for bit flips.

We ran the experiment for all five variations of AHEAD. The graphs in Figure 5.3 show the fault rate graphs for the normal, early onetime detection, late onetime detection, continuous detection and continuous detection with reencoding respectively. The prototype was modified in a way that when a bit flip is detected, the program only exits with a non-zero exit code. This means that the line that represents the *abnormal* behaviour actually represents bit flip detection.

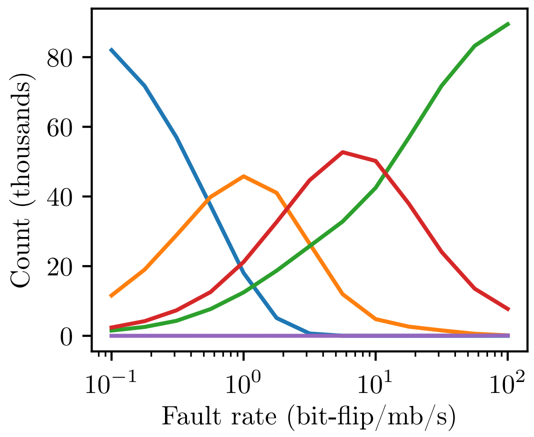
Running experiments on the normal variation yielded results similar to the results of the SQLite experiments. The rate of good results quickly plunges to zero. However, the rate of incorrect results is much higher reaching 66% at the fault rate of $10^{0.75}$. One of the reasons for this could be that AHEAD has the raw implementation of the queries with the physical operators. There is no query parsing, planning or optimisations phase where a bit flip would most likely crash the process. AHEAD also loads the entire tables into memory, making the sensitive data a large target for bit flips.



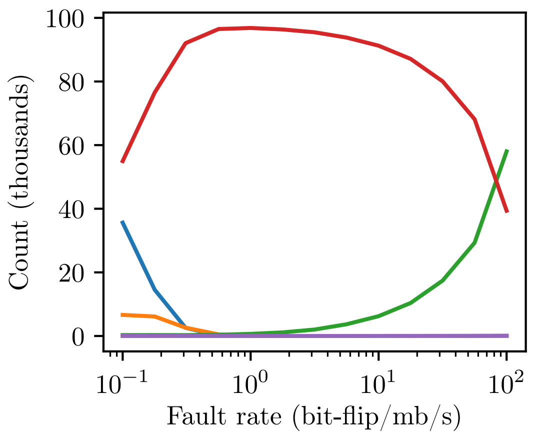
(a) Normal



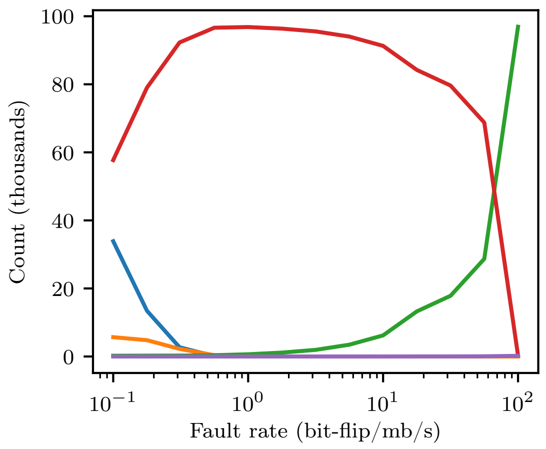
(b) Early detection



(c) Late detection



(d) Continuous detection



(e) Continuous detection with reencoding

🟦	ok
🟧	incorrect
🟩	crash
🟥	abnormal
🟪	timeout

(f) Legend

Figure 5.3: AHEAD results

The graph in Figure 5.3b for the early variant of AHEAD shows an surprising result. One would expect that since the detection happens in the beginning, the injected faults would have more chance at corrupting the data after loading the data into memory. A reasonable explanation would be that since AHEAD loads the entire tables into memory, most of the time will be spent on loading the data into memory. After that, the whole data is inside the memory, only small fraction of the query time is spent on the rest of the operators as they will be very fast in-memory calculations. Our experiments show that the chance of SDC is negligible and reaches 0% quickly. The rate of incorrect results start with 0.52% at a fault rate of 10^{-1} and quickly reaches 0% at the fault rate of $10^{0.75}$.

Experiments on the late variant also gives us unexpected results shown on Figure 5.3c. One would expect that late detection would catch more bit flips than early detection, but the results say otherwise. The rate of incorrect results still reach a very high value of 47% at the fault rate of 10^{0}. Although detected faults and crashes quickly take over at higher fault rates, it is unclear what the reason for this is without knowing the implementation details of AHEAD. In their work they do mention that the late variant *may* detect bit flips in the late stages of the execution plane. This can indicate that it works less efficiently than the other variants.

The continuous detection variant yielded similar results both with and without reencoding the data after each operator. The rate of SDCs is also pushed down to 0% on fault rates of $10^{0.25}$ and higher. However, they start at a slightly higher rate of 6.8% without reencoding and 5.8% with reencoding at the fault rate of 10^{-1} compared to early detection.

We observe that AHEAD performs considerable well at detecting bit flips in a chaotic environment with the exception of the late detection variant. AHEAD claims the that detection queries only takes 1.19 times longer on average then normal queries (24). This seems quite impressive considering it can prevent almost all of the silent data corruptions. However, AHEAD has several flaws that needs ironing out before it can be considered a final solution for query runtime bit flip detection.

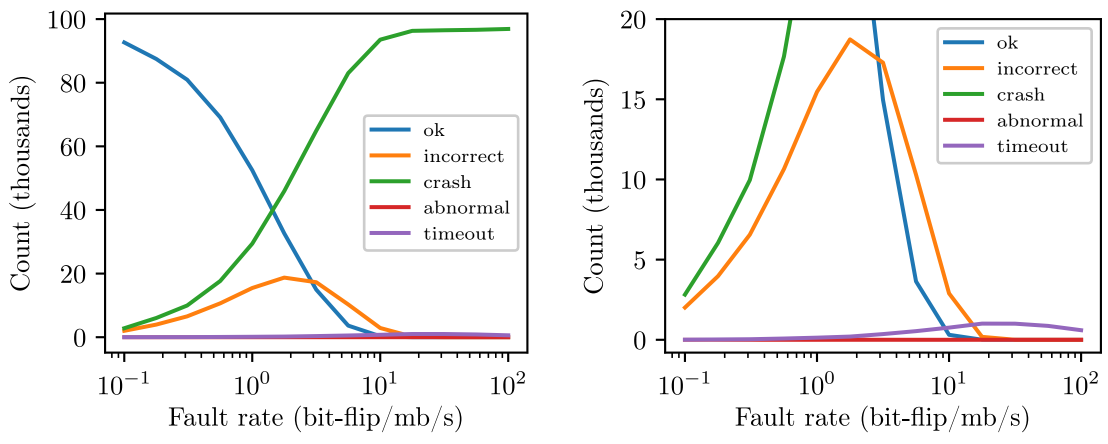
Firstly, AHEAD hardens data on disk and this hardening results in the storage usage being doubled. This can be a deal breaker for devices that do not have a lot of free storage. Secondly, AHEAD works with special operators that work on AN encoded data. Unless the system exclusively supports AN encoded data, all operators need to be implemented twice, making development and maintenance unnecessarily more tedious.

Although early detection performed really well, it also had the highest overhead and can be up to 50% slower than the normal variant. The two continuous modes also seemed to perform well. But if we take a closer look at the graphs, they performed just like the normal variant on lower fault rates, which is the more realistic scenario on healthy hardware. Finally, the AHEAD prototype has the Star Schema Benchmark queries hard coded without query parsing or query optimization. It also loads the entire tables into memory on start, therefore the performance benchmark results should be taken with a grain of salt.

Overall, AHEAD performs considerable well at detecting bit flips in a chaotic environment with the exception of the late detection variant. However, it still has several problems, that makes this solution immature for production systems.

5.3 DuckDB

Figure 5.4a shows the plot of the results of running TPC-H 1 queries on a DuckDB database. This plot of DuckDB results closely resembles the plot of SQLite results in Figure 5.1a, which is not surprising as both are embedded database systems. However, the main difference is in the probability of silent data corruptions. As it can be seen on Figure 5.4b the rate of incorrect result is a lot higher in case of DuckDB. The chance of getting an SDC reaches 19.2% at the fault rate of 1.778, which is almost 10 times higher than the probability of getting SDC from SQLite. While SQLite is designed to use as little memory as possible, DuckDB is designed for OLAP (Online Analytical Processing) workloads and can use more than 100 times the memory SQLite uses. This drastically increases the probability of a bit flip landing in parts of the memory where it can cause an SDC.



(a) All results

(b) Zoomed in

Figure 5.4: DuckDB results without protection

Even with those differences between SQLite and DuckDB, the similarities between the results of the experiments are quite interesting. It seems like in a rather chaotic environment, where we are flipping bits randomly, there is some kind of a system in the consequences. We expect this trend to continue when we move on to experimenting with other database systems.

Data accuracy is of key importance and OLAP databases are becoming increasingly important in data science. Data getting corrupted by a single bit could yield a wildly different result. As we have seen in Chapter 2, a surprisingly large amount of transient faults happen in hardware and the chance of such a fault causing a silent data corruption is fairly high. Since most consumer memory hardware comes without ECC, software level protection against bit flips would be necessary.

6

Mitigation Implementation

In this chapter we describe our prototype implementation of DuckDB that attempts to detect bit flips in memory during query execution. We present the results of the experiments performed on DuckDB prototypes that have a combination of these mitigation techniques implemented.

6.1 Intermediate Protection

DuckDB has a vectorized execution engine, which means that the physical operators process data in chunks. They operate with two basic units called DataChunks and Vectors. A vector contains a section of a column, a data chunk contains multiple columns as multiple vectors. These data chunks are created on the fly by the operators and are passed from operator to operator during a query execution making them vulnerable to bit flips.

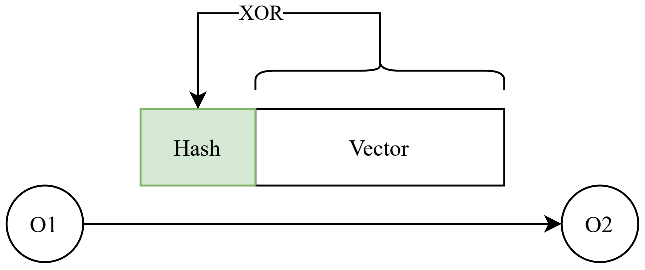


Figure 6.1: Intermediate protection

To protect detect unexpected changes in this intermediate data, we implemented a simple checksum verification on individual vectors. A simple diagram of this protection is shown on Figure 6.1. This checksum is computed by XOR-ing the values inside the vector together. When an operator has finished its computation on the chunks received by another operator, it verifies the checksum by recomputing it and checking against the first checksum value. If the verification fails, a bit flip is detected and the query execution is immediately stopped by throwing an exception.

6.2 Block Protection

The database of DuckDB is stored in a single file. Inside this file data is stored in blocks and the entire block is loaded into memory when data as read from it. A simple diagram of the structure of the block is shown on Figure 6.2. DuckDB already checks the integrity of the block after loading it into the memory. Every block has a header that contains a checksum value over the entire block. This checksum is verified upon loading the block into memory and if the verification fails, the data file is assumed corrupted and an exception is thrown.

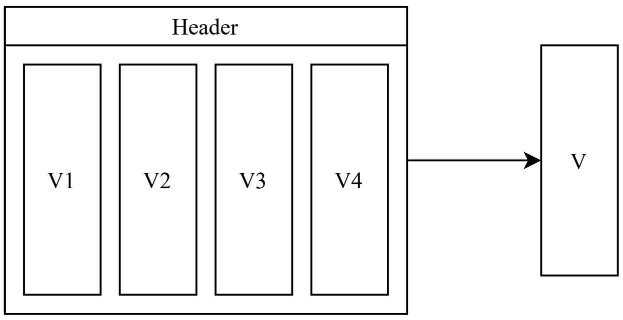


Figure 6.2: Block structure

We can take advantage of this checksum value and use it to detect bit flips inside entire blocks that reside in the main memory. The physical table scan operator reads vectors from these blocks. After reading data from the block we can redo the checksum verification to check whether the contents of the block changed or not. If the verification fails, we assume the block in the memory got corrupted and immediately stop the execution by throwing an exception.

6.3 Limitations and Performance Implications

The approach of intermediate protection has its limitation, some parts of the intermediate data are still not protected. The initial checksum calculation is not done before the operator has finished its job. The data chunks created by the operators are typically filled up with data gradually by the operator, making them vulnerable while they are being created. For a column of string data type, the vectors only contain pointers and the actual string values live in the string heap that belongs to the data chunk. When computing the checksum, only the values of the pointers are XOR-ed together, the strings themselves are not protected. Additionally, the hash table is also not protected even though it is a very important part of the query execution.

Naturally, this mitigation will have negative impacts on query performance. Currently we are mainly interested in the effect of these protections on the rate of silent data corruption. The implementation of reverification of the blocks is not optimized and we expect a major performance impact because of it. Blocks can contain multiple vectors and currently the checksum is reverified every time a single vector is read from the block. This can result in an undesirably large amount of checksum computations on the entire block. Additionally, the vectors are relatively small and we are checking the whole block just for reading a small part of it.

6.4 Results

We ran experiments on three variants of DuckDB that have a combination of protections enabled. One has only protection of intermediate data created by the operators. One has only block protection that is loaded into memory. Lastly, one has both intermediate and block protection. We ran the usual fault injection experiments on all variants. Additionally, we also measured the runtime of all variants on a number of TPC-H queries to asses how such mitigation techniques affect the performance of the database queries.

6.4.1 Performance

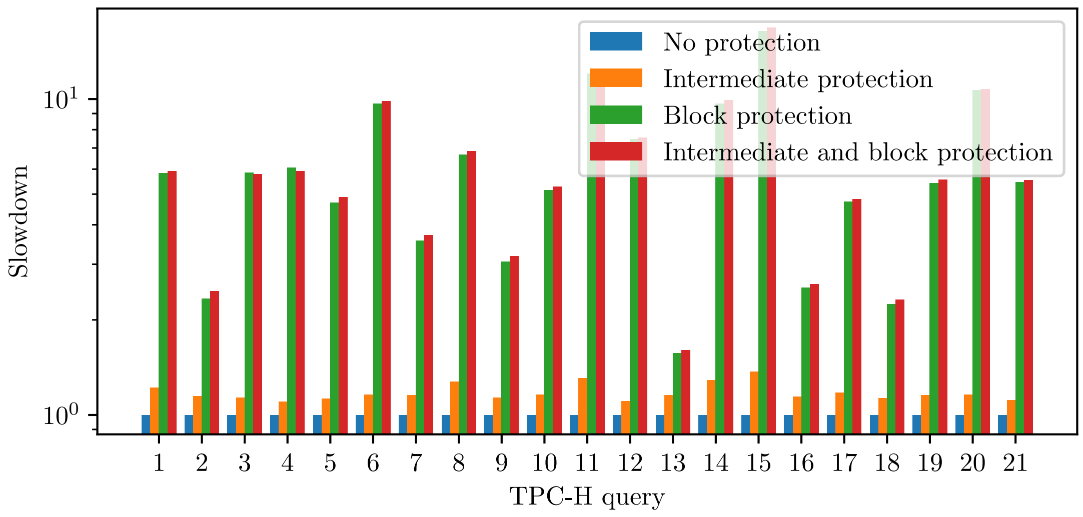
First we measured the performance of all the variants including the original version without fault protection implemented. We differentiate cold runs from hot runs. A cold run is the first query of a database system when first starting the process. It can take significantly longer as it needs to load data from persistent storage into memory or it needs to compile queries into native code. Subsequent queries are hot runs and are often faster as the operating system also caches the accessed data. Figure 6.3a shows the normalized run times of cold runs and Figure 6.3b shows the normalized run times of hot runs. The runtime axis has a logarithmic scale. On average, queries with intermediate protection only took 1.18 times longer for cold runs and 1.26 times longer for hot runs. Block protection had a significant impact on the performance, taking 6.24 times longer for cold runs and 9.16 times longer for hot runs to finish the query. The combination of the two protections added up in runtime. Cold runs for the combined variant took 6.36 times longer for cold runs and 9.36 times longer for hot runs on average.

The slight impact of intermediate protection is expected as XOR-ing values is a fast bit-wise operation without copying values around. However, the block protection has an undesirably big impact on the performance of queries. This is due to the inefficient implementation of the bit flip detection. Data is stored in blocks and each block can contain multiple vectors of data. The bit flip detection is performed every time a single vector is read from the block. This means that the whole block is scanned and the checksum is calculated for every vector that is read from the block. Although this implementation is expected to have a high coverage in space, both used and unused space in the memory is scanned multiple times, even though once would be enough. We will discuss possible improvements to block protection in Section 7. The main objective of these experiments is to show that simple checksumming over memory can significantly reduce the rate of silent data corruption.

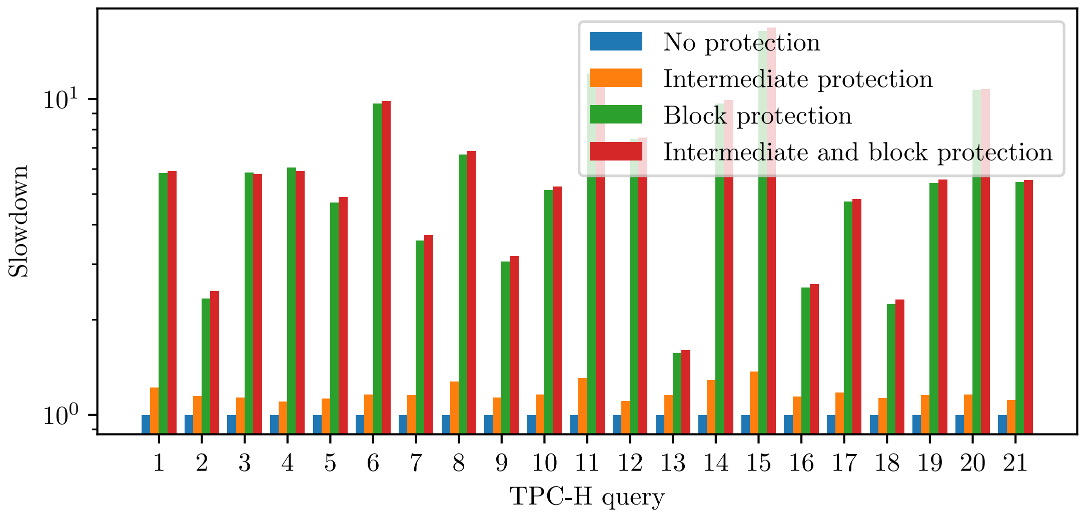
6.4.2 Fault-rate

When DuckDB detects a bit flip and exception is thrown. Normally, the database engine catches these exceptions and returns the error. This behaviour would mask the bit flip detection event and hide it from our testing framework. We have wrote our minimal program that uses DuckDB as a library and when an error is returned, it exits with a non-zero exit code. This abnormal behaviour with a non-zero exit code only happened twice out of almost a million runs with the variant that had the protections disabled. If we observe a this abnormal behaviour in later experiments with bit flip detection enabled, it will be highly likely caused by the detection. Similarly to AHEAD in Section 5.2, the lines indicating the abnormal behaviour in the graphs accurately represents the bit flip detection rate.

The fault rate graphs for the intermediate protection only variant of DuckDB is shown on Figure 6.4. We observe that, although a low amount, bit flips are indeed detected. This technique can detect memory faults in up to 6.2% of the queries. However, we find that on lower fault rates there are actually more incorrect results than in the unprotected version. We suspect that this is due to the slight increase in runtime due to the checksum calculations done by the operators. Since the query runs a little longer, slightly more bit flips are injected, giving data corruptions a higher chance to occur. As we increase the fault rate, more and more incorrect results are prevented until the crashes take over. As we have seen in the last section, this protection increases the runtime of queries by about

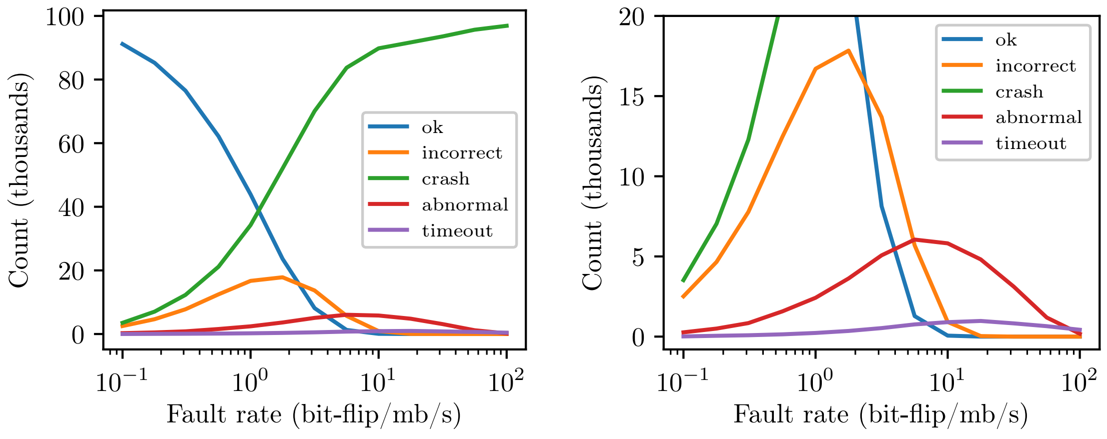


(a) Cold runs



(b) Hot runs

Figure 6.3: Benchmark of implemented mitigations using TPC-H



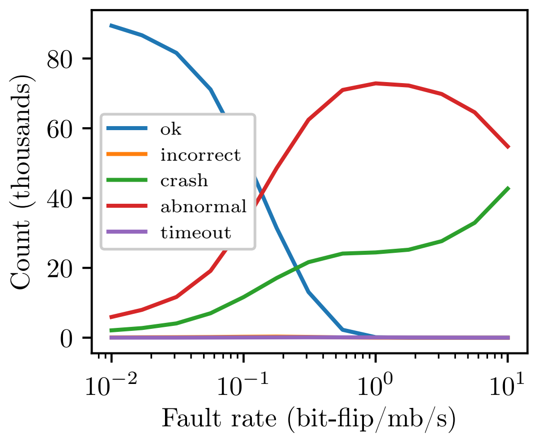
(a) All results **(b)** Zoomed in

Figure 6.4: DuckDB with intermediate protection only

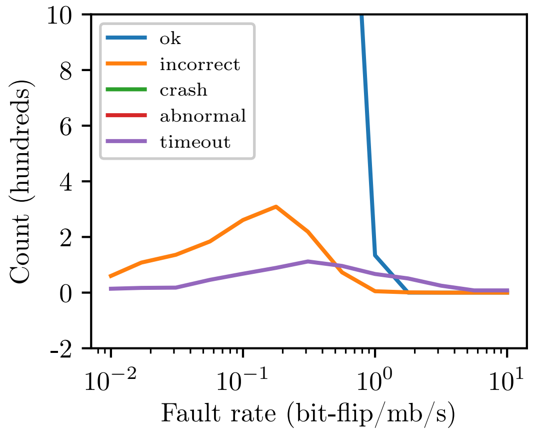
20%. This performance impact is quite large considering the small impact it has on the rate of SDCs.

The block protection variant performed much better than the intermediate protection did in terms of suppressing silent data corruptions. We can see on Figures 6.5a and 6.5b that the rate of detected bit flips out number even the crashes and the rate of SDCs is less than 0.32% which means that almost all of the queries detected a memory fault. However, block protection has a significant impact on performance as we have seen in section 6.4.1, therefore in order to better measure the percentage of suppressed incorrect results we also ran experiments on a version of DuckDB that has the protections implemented, but exceptions are not thrown when bit flips are detected. This gives us a better comparison between a protected and an unprotected version of DuckDB as they will have the same performance. The fault rate graphs for this variant is shown in Figure 6.6. The graphs look quite similar to graphs of the normal DuckDB in Figure 5.4. Note that the X axis of the graphs for variants that include block protection start at 10^{-2} instead of 10^{-1} because the queries in these experiments take a long time to finish. This means more bit flips for each fault rate visually shifting the graph to the left along the X axis. once we take a look at the exact numbers, we can observe that block protection reduces the rate of SDCs to almost zero.

Finally, we ran experiments on DuckDB that has both intermediate and block protection implemented. Results are shown in Figure 6.7. The graph looks near identical to previous one with minor differences. Adding intermediate protection to block protection

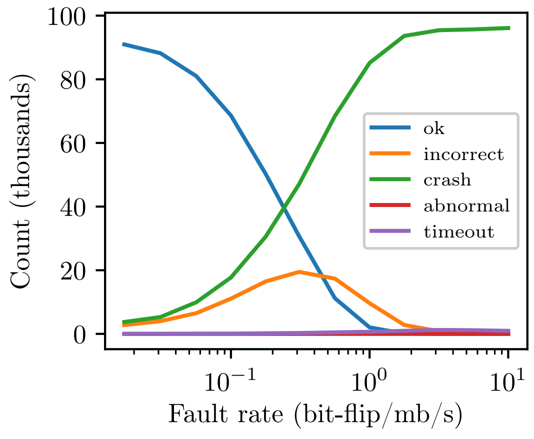


(a) All results

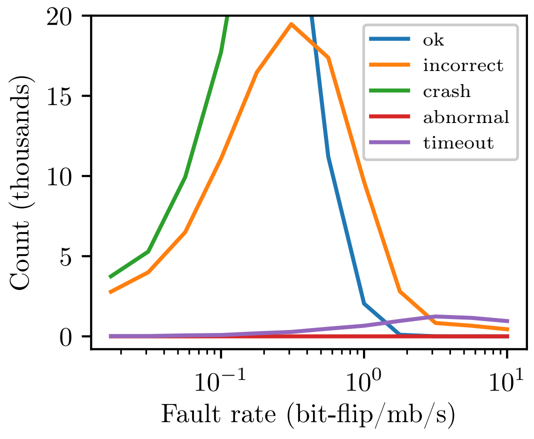


(b) Zoomed in

Figure 6.5: DuckDB with block protection only

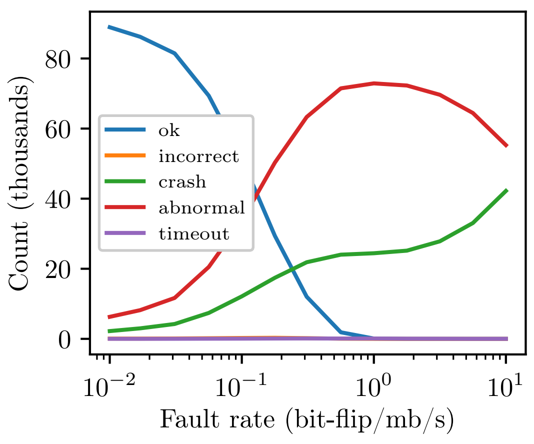


(a) All results

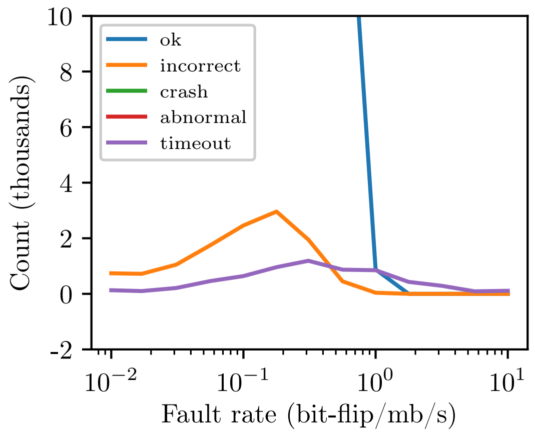


(b) Zoomed in

Figure 6.6: DuckDB with block protection without raising errors



(a) All results



(b) Zoomed in

Figure 6.7: DuckDB with both intermediate and block protection

has a relatively small effect on an already low rate of SDCs, which is to be expected after experimenting on intermediate protection only.

The small impact of intermediate protection on SDC rates can be explained by the fact that the operators usually perform simple operations on the vectors. These vectors are also rather small making them short lived as they are quickly deallocated once the operator is done. Considering this, the probability of a bit flip hitting a vector in memory and the fault actually getting detected is very small, because the given space and time frame is equally very small. On the other hand, protecting the blocks that contain these vectors has a significant improvement on SDC rates almost completely suppressing all incorrect results. This is likely due to the blocks being larger in size compared to vectors, making them a big target for bit flips. Unlike intermediate data chunks, blocks are also retained in memory for longer as multiple vectors can be read from them throughout the lifetime of the database connection.

In conclusion, we observe that protection against bit flips in main memory for databases is certainly possible. Block protection prevented silent data corruptions in almost all of the queries while intermediate protection only prevented them in a fraction of the samples. However, the runtime performance of block protection is terrible, and performs a large amount of unnecessary computation. There are many ideas on how we could improve the performance of block protection. These possible improvements are discussed in the next chapter.

7

Future Work

7.1 Fault injector

Our fault injector is currently a rather simple platform independent program that only supports the injection of single bit flips in the main memory. This is quite far from covering the wide range of faults that can happen in hardware. Bit flips are transient faults, however, some of the studies have shown that corrupted memory content is dominantly caused by hard faults(17, 27, 32), such as stuck bits. Although soft and hard faults are effectively the same for single reads, it would not reflect reality to simulate hard faults with simple bit flips. We suspect that it is not possible with ptrace to stick a bit at an address to a constant 0 or 1. It may be possible to emulate hard or intermittent faults using the single step mode of a CPU(41). However, single step mode is incredibly slow, making large scale experiments like the ones we conducted not viable.

Additionally, the fault injector only supports one type of fault in terms of affected space. The injected bit flips only happen to bits in the memory that are independent from each other. A fault can affect entire words, rows, columns or banks of a memory hardware. Linux systems utilize virtual addresses, that are translated from physical addresses to make memory management easier for user software. Without knowing how this translation happens, it is impossible to simulate faults like row, column, etc. hardware faults. The area of injecting hard faults requires more investigation and will likely require an interface or tool different from ptrace.

The main memory is not the only place where data can reside. Bit flips can also happen on disk, but it is easier to protect data on disk against soft faults. File checksums are a widely used technique for ensuring file integrity and it is already utilized by DuckDB. Moreover, the RAID (Redundant Array of Independent Disks) technology can be used to apply redundancy to stored data, to protect against unexpected disk failures. However, bit flips can also happen in other components of a computer, such as CPU registers, the CPU cache or FPUs. The ptrace interface can give us access to the registers of the traced process, but this is not present on all Linux architectures. CPU caches work transparently and are architecture dependent. Some CPU architectures might provide debugging interfaces to inspect at modify the CPU cache. This kind of fine grained access to this CPU cache would probably require elevated permission, which potentially negatively impacts the viability of large scale experiments on clusters.

When using the ptrace system call, the kernel acts as a middleman between the tracer and the tracee. This doubles the number of context switches compared to usual system calls causing a significant overhead. Additionally, while the process is attached with ptrace, the child execution of the child process is stopped, further increasing a runtime of a query during an experiment. This overhead could potentially be avoided by using the `process_vm_readv(2)` and the `process_vm_writev(2)`[1] system calls to modify the memory of another process. Although ptrace is still required for the permission to modify the memory content, this method does not stop the execution of the tracee. This could give a noticeable performance boost when running experiments millions of times.

Lastly, our tool currently only supports injecting faults into single process database systems, like SQLite and DuckDB. We implemented support for single process server based database systems, but ultimately yielded invalid when we ran our experiments on MonetDB. We suspect that it is caused by some kind of a bug in the injector and the way MonetDB allocates memory, and it would need further investigation. More complex systems may have their own child processes and these processes may or may not have shared memories. It is theoretically possible to inject faults into systems like this as it is possible to build the child process tree and keep track of what address spaces are used by each process.

7.2 Mitigation

We implemented a prototype of DuckDB with certain protections against silent data corruptions caused by bit flips in memory. We believe we achieved promising results, but our approach could be further improved in several ways.

Although small percent, there are still parts of the memory that is not protected during a query execution. The vectors created by operators are only protected after the operator

[1]http://man7.org/linux/man-pages/man2/process_vm_readv.2.html

has finished, they are not protected while they are being slowly filled with data. Some would consider the chance of a bit flip happening during vector creation to be negligible. However it is not zero, and protection against it must be considered. Additionally, strings are stored in the string heap and the vectors only store the pointers to a location in the heap. This string heap is not currently protected. Strings are a data type that take up a relatively large chunk of the memory making them a big target for bit flips. Only the pointers to the strings inside the vectors are protected against bit flips. Lastly, the hash table, which is a rather important part of a query execution, is also not protected. Hash tables can be bigger than other intermediates and live longer than the vectors created by the operators.

We have seen in section 6.4.1 that the performance of our implementation of block protection is quite poor with TPC-H queries taking many times longer than normal queries. For database systems this is unacceptable as one of the key requirements users tend to have is fast query performance. We have several ideas how it could be optimized for better performance. One possibility to protect data in the blocks is to utilize a similar technique we used for protecting intermediate vector data. DuckDB could additionally store the vector checksum values inside the blocks and when they are read from the block we only need to calculate the checksum for the vector instead of doing it for the whole block. We expect this approach to detect less bit flips overall but suppress SDCs equally well.

Other possible methods include storing a hash tree in the header of the block instead of a single hash for the entire block. The hash tree would contain multiple hashes and each hash would correspond to different parts of the block. And when we read a vector from the block we only need to calculate a smaller number of hashes again for the parts of the block that were touched. This approach has the benefit of not having to modify the binary format of the file to add headers to the stored vectors. Similarly to the previous method, we expect this one to improve the query performance and prevent SDCs just as efficiently.

Lastly, a simpler method would be to only run the checksum calculation after multiple reads to the block. This can be considered as late bit flip detection, since a corrupted memory could be read and used multiple times before the software has a chance to detect the error. While this method offers a much better performance than our current implementation, this is not an ideal solution as the bit flip has a higher chance of causing a software failure and can go undetected.

These are only our initial ideas for improving query performance while maintaining good fault protection performance. With continuous research in this largely unexplored area, we believe that it is possible to achieve better and better performance over time.

8

Conclusion

The practice of testing resiliency with fault injection has been around for a while(30). Netflix took this concept further and defined four principles of chaos engineering they use to run fault tolerance tests on their production system(5). We found that three out of the four principles can easily be adapted to run fault tolerance experiments on database systems. The only problematic part is running tolerance test continuously. In our experiments we need to run queries millions of time, which takes a long time even on a cluster.

There have been several studies on hardware memory faults in the field (17, 25, 26, 31, 32, 33, 38, 40, 43). From these surveys we concluded that memory faults happen more often than one would expect. Additionally these studies agree that this issue is slowly becoming bigger with devices coming with more and more memory equipped and the world depending on more and more data. There is a need to understand the nature of these faults and their effect on software applications and services.

Therefore we need to somehow simulate a chaotic environment with bit flips and run the database systems in it. The existing fault injection tools did not meet our requirements so we implemented our own using the ptrace Linux interface. We used our injection tool to perform experiments on SQLite, DuckDB and AHEAD. We could already tell that SQLite and DuckDB can be quite vulnerable to silent data corruptions. SQLite yielded a staggeringly high chance of getting the database file itself corrupted if the bit flip happens during an update query. We could conclude from these baseline experiments, that these databases do not handle chaotic environments well. The AHEAD column store prototype claims to be able to detect bit flips during query execution time with minimal overhead. We ran our experiments on AHEAD and although the prototype had some promising results in our experiments, there is still quite a lot of room for improvement.

8. CONCLUSION

We implemented two simple techniques into DuckDB to detected bit flips during query execution time. While intermediate protection only detected a fraction of the injected bit flips, block detected managed to prevent more than 96% of all silent data corruptions. However, block detection has poor performance in its current implementation, but we already have several ideas on how to improve it. And compared to AHEAD, which has other drawbacks, performance is the only negative side effect of our implementation. We believe that block detection could be a valuable addition to analytical databases, where the incorrectness of data is unacceptable.

We now have a better understanding of how transient faults can affect database systems. We know that it is possible to prevent silent data corruptions in databases on a software level so it can be used on commodity systems as well. We believe that we have established a good basis for database chaos engineering. The area of experimenting with fault tolerance in databases unexplored and we have seen in Section 7 that there are still a lot of work to in our study.

9

Appendix

9.1 Literature Study

Our work involves injects main memory faults into database systems and protecting against these faults. Since this is quite a specific and not well explored area of study, there are only a few directly related works. We defined to following broader **research question** to define our topic of interest and identify interesting publications: The prominence of hardware memory faults, their affect on database software and possible techniques for mitigation.

We initially search for papers using keywords and phrases such as chaos engineering, fault injection, DRAM faults, bit flips, databases and silent data corruption. From then on we explore additional works that are referenced in the papers we have already found. All the papers are organized into a large table. Table 9.2 represents that table with a subset of the columns. To help with the organization, we classified every paper into one of the four categories shown in Table 9.1.

Although we found 130 papers, not all of them were relevant enough or some of them had an overlap in the contained information. To filter out the papers and find the most relevant ones, we defined several inclusion criterion shown in Table 9.2. These criterion are not hard requirements, their aim is to help us decide whether we should include a study in our literature study or not. Papers included in the literature study marked by an X in the column named S. in Table 9.2.

Categories	
Chaos Engineering (ce)	Studies presenting the chaos engineering concept or the application of chaos engineering in the field.
Fault Study (fs)	Studies presenting surveys on DRAM faults in large scale environments and emphasize the importance of dealing with such faults.
Fault Injection (fi)	Studies presenting solutions for fault injection based testing for fault tolerance evaluation.
Resiliency (r)	Studies that present various solutions for implementing of hardware fault resiliency on different platforms.

Table 9.1: Categories

Inclusion criteria		**Rationale**
1	A study that that proposes DRAM fault resiliency solutions for databases.	We are interested in papers closely related to our research, that we can compare our approach with. We can also possibly apply our experiment to these solutions.
2	A study that contains relevant and useful information for our thesis.	Studies that are related to our work might contain information that is redundant or irrelevant for our work.
3	A study that falls into at least on of the categories listed above.	The area of hardware faults in databases is largely unexplored. We are interested in these categories that are related to parts of our research.
4	A study that is developed by either academics or practitioners.	Both academic and industrial solutions are relevant to this study.
5	A study that is written in English.	For feasibility reasons, papers written in other languages than English are excluded.

Table 9.2: Inclusion criterias

#	**Title**	**Author**	**Year**	**Cat.**	**S.**	**Inclusion**				
						1	2	3	4	5
1	Chaos Engineering	Ali Basiri et al.	2016	ce	x		x	x	x	x
2	Chaos Monkey Increasing SDN Reliability through Systematic Network Destruction	Michael Alan Chang et al.	2015	ce				x	x	x
3	Chaos engineering and its application to parallel distributed processing with chaotic neural networks	Kazuyuki Aihara	2002						x	x
4	Automating Failure Testing Research at Internet Scale	Peter Alvaro et al.	2016	ce				x	x	x
5	A Platform for Automating Chaos Experiments	Ali Basiri et al.	2016	ce	x		x	x	x	x

#	Title	Author	Year	Cat.	S.	Inclusion				
						1	2	3	4	5
6	A realistic evaluation of memory hardware errors and software system susceptibility	Xin Li et al.	2010	fs	x		x	x	x	x
7	Increasing relevance of memory hardware errors: a case for recoverable programming models	Dejan Milojicic et al.	2000	fs				x	x	x
8	DRAM errors in the wild: a large-scale field study	Bianca Schroeder et al.	2009	fs	x		x	x	x	x
9	Resilience Engineering: Learning to Embrace Failure	Jesse Robbins et al.	2012	fi				x		x
10	Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems	Ding Yuan et al.	2014						x	x
11	Fault Injection Techniques - A Brief Review	Rakesh Kumar Lenka et al.	2018	fi				x	x	x
12	Automating Chaos Experiments in Production	Ali Basiri et al.	2019	ce	x		x	x	x	x
14	Designing reliable systems from unreliable components: the challenges of transistor variability and degradation	Shekhar Borkar	2005	r					x	x
15	Do we need anything more than single bit error correction (ECC)?	Michael Spica et al.	2004	r				x	x	x
16	Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design	Andy A. Hwang et al.	2012	fs	x		x	x	x	x
17	Reliable Software for Unreliable Hardware - A Cross Layer Perspective	Semeen Rehman	2016	r				x	x	x
18	Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors	Yoongu Kim et al.	2014						x	x
19	Memory Errors in Modern Systems: The Good, The Bad, and The Ugly	Vilas Sridharan et al.	2015	fs				x	x	x
20	A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM	Samira Khan et al.	2017	r				x	x	x
21	Detecting and mitigating data-dependent DRAM failures by exploiting current memory content	Samira Khan et al.	2017	r				x	x	x
22	Online bit flip detection for in-memory B-trees on unreliable hardware	Till Kolditz et al.	2014	r		x		x	x	x

#	Title	Author	Year	Cat.	S.	Inclusion				
						1	2	3	4	5
23	Fault injection techniques and tools	Mei-Chen Hsueh et al.	1997	fi	x		x	x	x	x
24	Xception: a technique for the experimental evaluation of dependability in modern computers	João Carreira et al.	1998	fi	x		x	x	x	x
25	Understanding the propagation of hard errors to software and implications for resilient system design	Man-Lap Li et al.	2008	fs	x		x	x	x	x
26	Comparison of physical and software-implemented fault injection techniques	Jean Arlat et al.	2003	fi				x	x	x
27	Definition and analysis of hardware- and software-fault-tolerant architectures	Jean-Claude Laprie et al.	1990	fs				x	x	x
28	Software-Implemented Hardware Fault Tolerance	Olga Goloubeva et al.	2006	r				x	x	x
29	Fault injection tools based on Virtual Machines	Maha Kooli et al.	2014	fi				x	x	x
30	Fast Simulation of Stuck-At and Coupling Memory Faults Using FAUmachine	Hans-Jörg Höxer et al.	2005	fi				x	x	x
31	Evaluating fault-tolerant system designs using FAUmachine	Stefan Potyra et al.	2007	fi				x	x	x
32	MAFALDA: Microkernel Assessment by Fault Injection and Design Aid	Manuel Rodríguez et al.	2000	fi	x		x	x	x	x
33	Fault injection experiments using FIAT	Manuel Rodríguez et al.	1990	fi	x		x	x	x	x
34	Xception: Software Fault Injection and Monitoring in Processor Functional Units	João Carreira et al.	1998	fi				x	x	x
35	FERRARI: a tool for the validation of system dependability properties	Ghani A. Kanawati et al.	1992	fi	x			x	x	x
36	Improving the reliability of commodity operating systems	Michael M. Swift	2003	r				x	x	x
37	Fast byte-granularity software fault isolation	Miguel Castro et al.	2009	r				x	x	x
38	Protecting Commodity Operating System Kernels from Vulnerable Device Drivers	Shakeel Butt et al.	2009	r				x	x	x
39	Building a Self-Healing Operating System	Francis M. David et al.	2007	r				x	x	x
40	CuriOS: Improving Reliability through Operating System Structure	Francis M. David et al.	2008	r				x	x	x

#	Title	Author	Year	Cat.	S.	Inclusion				
						1	2	3	4	5
41	Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation	João Durães	2002	r				x	x	x
42	Why Do Computers Stop and What Can Be Done About It?	Jim Gray	1985						x	x
43	High-availability computer systems	Jim Gray et al.	1991						x	x
44	Tolerating hardware device failures in software	Asim Kadab et al.	2009	r				x	x	x
45	FERRARI: a flexible software-based fault and error injection system	Ghani A. Kanawati et al.	1995	fi				x	x	x
46	An Approach to Designing Fault-Tolerant Computing Systems	Richard D. Schlichting et al.	1981	fi	x		x	x	x	x
47	Ensuring data integrity in storage: techniques and applications	Gopalan Sivathanu et al.	2005	r				x	x	x
48	Construction of a Highly Dependable Operating System	Jorrit N. Herder et al.	2006	r				x	x	x
49	Failure Resilience for Device Drivers	Jorrit N. Herder et al.	2007	r				x	x	x
50	Dealing with Driver Failures in the Storage Stack	Jorrit N. Herder et al.	2009	r				x	x	x
51	A Survey on Fault Injection Techniques	Haissam Ziade et al.	2004	fi	x		x	x	x	x
52	Fault injection for dependability validation: a methodology and some applications	Jean Arlat et al.	1990	fi				x	x	x
53	AHEAD: Adaptable Data Hardening for On-the-Fly Hardware Error Detection during Database Query Processing	Till Kolditz et al.	2018	r	x	x	x	x	x	x
54	Needles in the Haystack âĂŤ Tackling Bit Flips in Lightweight Compressed Data	Till Kolditz et al.	2016	r	x	x	x	x	x	x
55	Resiliency-aware Data Compression for In-memory Database Systems	Till Kolditz et al.	2015	r	x	x	x	x	x	x
56	Resiliency-Aware Data Management	Till Kolditz et al.	2011	r	x	x	x	x	x	x
57	Using memory errors to attack a virtual machine	Sudhakar Govindavajhala et al.	2003						x	x
58	Susceptibility of commodity systems and software to memory soft errors	Alan Messer et al.	2004	fs	x		x	x	x	x
59	Protecting critical data in unsafe languages	Karthik Pattabiraman et al.	2008	r				x	x	x

#	Title	Author	Year	Cat.	S.	Inclusion				
						1	2	3	4	5
60	The Rio File Cache: Surviving Operating System Crashes	Peter M. Chen et al.	1996	r				x	x	x
61	Redundancy in Data Structures: Improving Software Fault Tolerance	David J. Taylor et al.	1980	r				x	x	x
62	Large Scale Studies of Memory, Storage, and Network Failures in a Modern Data Center	Justin Meza	2019	fs				x	x	x
63	Revisiting memory errors in large-scale production data centers: analysis and modeling of new trends from the field	Justin Meza et al.	2015	fs	x		x	x	x	x
64	A large-scale study of flash memory errors in the field	Justin Meza et al.	2015	fs				x	x	x
65	A study of DRAM failures in the field	Vilas Sridharan et al.	2012	fs				x	x	x
66	Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults	Vilas Sridharan et al.	2013	fs	x		x	x	x	x
67	Extra bits on SRAM and DRAM errors - more data from the field	Nathan DeBardeleben et al.	2014	fs				x	x	x
68	Analysis and Modeling of Memory Errors from Large-scale Field Data Collection	Taniya Siddiqua et al.	2013	fs				x	x	x
69	Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs	Edmund B. Nightingale et al.	2011	fs	x		x	x	x	x
70	A Memory Soft Error Measurement on Production Systems	Xin Li et al.	2007	fs	x		x	x	x	x
71	What can we learn from four years of data center hardware failures?	Guosai Wang et al.	2017	fs				x	x	x
72	Lifetime memory reliability data from the field	et al.	2017	fs	x		x	x	x	x
73	Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors	Weining Gu et al.	2004						x	x
74	Error Detection Using Dynamic Dataflow Verification	Albert Meixner et al.	2007	r				x	x	x
75	Virtualized and flexible ECC for main memory	Doe Hyun Yoon et al.	2010	r				x	x	x
76	Software Design for Resilient Computer Systems	Igor Schagaev et al.	2016	r				x	x	x

#	**Title**	**Author**	**Year**	**Cat.**	**S.**	**Inclusion**				
						1	2	3	4	5
77	Cooperative Application/OS DRAM Fault Recovery	Patrick G. Bridges et al.	2011	r				x	x	x
78	Generative software-based memory error detection and correction for operating system data structures	Christoph Borchert et al.	2013	r				x	x	x
79	SystemC-Based Minimum Intrusive Fault Injection Technique with Improved Fault Representation	Rishad Ahmed Shafik et al.	2008	fi				x	x	x
80	Detection and correction of silent data corruption for large-scale high-performance computing	David Fiala	2012	r				x	x	x
81	Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory	Jungrae Kim et al.	2015	r				x	x	x
82	Low-power, low-storage-overhead chipkill correct via multi-line error correction	Xun Jian et al.	2013	r	x		x	x	x	x
83	Quantifying the Impact of Single Bit Flips on Floating Point Arithmetic	James Elliott et al.	2013	fs				x	x	x
84	Improving DRAM Fault Characterization through Machine Learning	Elisabeth Baseman et al.	2016	fs				x	x	x
85	Unprotected computing: a large-scale study of DRAM raw error rate on a supercomputer	Leonardo Bautista-Gomez et al.	2016	fs				x	x	x
86	Measuring the Impact of Memory Errors on Application Performance	Mark Gottscho et al.	2016	fs				x	x	x
87	Software-defined error-correcting codes	Mark Gottscho et al.	2016	r				x	x	x
88	Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory	Yixin Luo et al.	2014						x	x
89	Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool	Dong Li et al.	2012	fi				x	x	x
90	Application-Level Correctness and its Impact on Fault Tolerance	Xuanhua Li et al.	2007						x	x
91	Fault injection framework for system resilience evaluation: fake faults for finding future failures	Thomas Naughton et al.	2009	fi				x	x	x
92	Soft error vulnerability of iterative linear algebra methods	Greg Bronevetsky et al.	2008	fi				x	x	x

#	Title	Author	Year	Cat.	S.	Inclusion				
						1	2	3	4	5
93	Experimental Framework for Injecting Logic Errors in a Virtual Machine to Profile Applications for Soft Error Resilience	Nathan DeBardeleben et al.	2011	fi				x	x	x
94	Assessing fault sensitivity in MPI applications	Charng-da Lu et al.	2004	fi				x	x	x
95	Characterizing the impact of soft errors on iterative methods in scientific computing	Manu Shantharam et al.	2011	fi				x	x	x
96	Analyzing the soft error resilience of linear solvers on multicore multiprocessors	Konrad Malkowski et al.	2010	fi				x	x	x
97	A framework for efficiently analyzing architecture-level fault tolerance behavior in applications	Harshad Sane	2008	fi				x	x	x
98	Algorithm-based fault tolerance for matrix operations	Kuang-Hua Huang et al.	1984	r				x	x	x
99	Algorithm-based recovery for iterative methods without checkpointing	Zizhong Chen	2011	r				x	x	x
100	Algorithm-based fault tolerance for dense matrix factorizations	Peng Du et al.	2012	r				x	x	x
101	Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance	Alex Shye et al.	1976	r				x	x	x
102	Evaluating the viability of process replication reliability for exascale systems	Kurt Ferreira et al.	2011	r				x	x	x
103	Design, modeling, and evaluation of a scalable multi-level checkpointing system	Adam Moody et al.	2010	r				x	x	x
104	A Case for Multi-Level Distributed Recovery Schemes	Nitin H. Vaidya	1994	r				x	x	x
105	A model of roll-back recovery with multiple checkpoints	Erol Gelenbe	1976	r				x	x	x
106	DOCTOR: an integrated software fault injection environment for distributed real-time systems	Seungjae Han et al.	1993	fi				x	x	x
107	EXFI: a low-cost fault injection system for embedded microprocessor-based boards	Alfredo Benso et al.	1998	fi	x		x	x	x	x
108	NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors	David T. Stott et al.	2000	fi	x		x	x	x	x
109	GOOFI-2: A tool for experimental dependability assessment	Daniel Skarin et al.	2010	fi	x		x	x	x	x

#	**Title**	**Author**	**Year**	**Cat.**	**S.**	**Inclusion**				
						1	2	3	4	5
110	GOOFI: generic object-oriented fault injection tool	Jan Aidemark et al.	2001	fi	x		x	x	x	x
111	Assessing Dependability with Software Fault Injection: A Survey	Roberto Natella et al.	2016	fi				x	x	x
112	ORCHESTRA: a probing and fault injection environment for testing protocol implementations	Scott Dawson et al.	1996	fi				x	x	x
113	FIG: A Prototype Tool for Online Verification of Recovery Mechanisms	Pete Broadwell et al.	2002	fi				x	x	x
114	Fault-Injector using UNIX ptrace Interface	Volkmar Sieh et al.	1993	fi			x	x	x	x
115	Framework for testing the fault-tolerance of systems including OS and network aspects	Kerstin Buchacker et al.	2001	fi				x	x	x
116	Plug and Play Fault Injector for Dependability Benchmarking	Pedro Costa et al.	2003	fi				x	x	x
117	Alpha-particle-induced soft errors in dynamic memories	Timothy C. May et al.	1979	fs	x		x	x	x	x
118	Single event upset at ground level	Eugene Normand et al.	1996	fs	x		x	x	x	x
119	Soft errors in advanced computer systems	Robert Baumann et al.	2005	fs	x		x	x	x	x
120	Scaling and technology issues for soft error rates	Allan H. Johnston	2000	fs	x		x	x	x	x
121	IBM experiments in soft fails in computer electronics (1978-1994)	James F. Ziegler et al.	1996	fs	x		x	x	x	x
122	Cosmic Ray Soft Error Rates of 16-Mb DRAM Memory Chips	James F. Ziegler et al.	1998	fs	x		x	x	x	x
123	A white paper on the benefits of chipkill-correct ECC for PC server main memory	Timothy J. Dell	1997	r	x		x	x	x	x
124	RIFLE: A general purpose pin-level fault injector	Henrique Madeira et al.	1994	fi				x	x	x
125	FOCUS: An experimental environment for fault sensitivity analysis	G.S. Choi et al.	1992	fi	x		x	x	x	x
126	Fault injection for dependability validation of fault-tolerant computing systems	Jean Arlat et al.	1989	fi	x		x	x	x	x
127	Using heavy-ion radiation to validate fault-handling mechanisms	Johan Karlsson et al.	1994	fi	x		x	x	x	x
128	An evaluation of the error detection mechanisms in MARS using software-implemented fault injection	Emmerich Fuchs	1996	fi				x	x	x

#	Title	Author	Year	Cat.	S.	Inclusion				
						1	2	3	4	5
129	Resiliency Mechanisms for In-Memory Column Stores	Till Kolditz et al.	2018	r		x		x	x	x
130	New TPC benchmarks for decision support and web commerce	Meikel Poess et al.	2000						x	x

Table 9.3: Related papers

References

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. **GOOFI: generic object-oriented fault injection tool**. In *2001 International Conference on Dependable Systems and Networks*, pages 83–88, July 2001. 8

[2] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. **Automating Failure Testing Research at Internet Scale**. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 17–28, New York, NY, USA, 2016. ACM. Available from: `http://doi.acm.org/10.1145/2987550.2987555`. 1

[3] J. Arlat, Y. Crouzet, and J. . Laprie. **Fault injection for dependability validation of fault-tolerant computing systems**. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 348–355, June 1989. 7

[4] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. **Fault injection experiments using FIAT**. *IEEE Transactions on Computers*, **39**(4):575–582, April 1990. 8

[5] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. **Chaos Engineering**. *IEEE Software*, **33**(3):35–41, 5 2016. 1, 5, 43

[6] A. Basiri, L. Hochstein, N. Jones, and H. Tucker. **Automating Chaos Experiments in Production**. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 31–40, 5 2019. 5

[7] R. Baumann. **Soft errors in advanced computer systems**. *IEEE Design Test of Computers*, **22**(3):258–266, May 2005. 4

[8] A. Benso, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. **EXFI: A Low-cost Fault Injection System for Embedded Microprocessor-based Boards**. *ACM Trans. Des. Autom. Electron. Syst.*, **3**(4):626–634, October 1998. Available from: `http://doi.acm.org/10.1145/296333.296351`. 8

[9] Netflix Technology Blog. **The Netflix Simian Army**, 2011. Available from: `https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116` [cited 2019-09-05]. 1, 5

[10] A. Blohowiak, A. Basiri, L. Hochstein, and C. Rosenthal. **A Platform for Automating Chaos Experiments**. In *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 5–8, 10 2016. 5

[11] Matthias Böhm, Wolfgang Lehner, and Christof Fetzer. **Resiliency-Aware Data Management**. *Proceedings of the VLDB Endowment*, **4**(12), 2011. 7

[12] J. Carreira, H. Madeira, and J. G. Silva. **Xception: a technique for the experimental evaluation of dependability in modern computers**. *IEEE Transactions on Software Engineering*, **24**(2):125–136, Feb 1998. 8

[13] G. Choi and R. Iyer. **FOCUS: An Experimental Environment for Fault Sensitivity Analysis**. *IEEE Transactions on Computers*, **41**(12):1515–1526, dec 1992. 7

[14] Timothy J Dell. **A white paper on the benefits of chipkill-correct ECC for PC server main memory**. *IBM Microelectronics Division*, **11**:1–23, 1997. 5, 7

[15] Richard Hipp. **Most Widely Deployed and Used Database Engine**, 2019. Available from: `https://www.sqlite.org/famous.html` [cited 2019-11-15]. 17

[16] Richard Hipp. **Well-Known Users of SQLite**, 2019. Available from: `https://www.sqlite.org/mostdeployed.html` [cited 2019-11-26]. 18

[17] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. **Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design**. *SIGPLAN Not.*, **47**(4):111–122, March 2012. Available from: `http://doi.acm.org/10.1145/2248487.2150989`. 1, 4, 5, 39, 43

[18] Xun Jian, Henry Duwe, John Sartori, Vilas Sridharan, and Rakesh Kumar. **Low-power, Low-storage-overhead Chipkill Correct via Multi-line Error Correction**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 24:1–24:12, New York, NY, USA, 2013. ACM. Available from: `http://doi.acm.org/10.1145/2503210.2503243`. 7

[19] Allan H Johnston. **Scaling and technology issues for soft error rates**. 2000. 4

[20] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. **FERRARI: a tool for the validation of system dependability properties**. In *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 336–344, 7 1992. 8

[21] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. **Using heavy-ion radiation to validate fault-handling mechanisms**. *IEEE Micro*, **14**(1):8–23, Feb 1994. 7

[22] Till Kolditz, Dirk Habich, Patrick Damme, Wolfgang Lehner, Dmitrii Kuvaiskii, Oleksii Oleksenko, and Christof Fetzer. **Resiliency-Aware Data Compression for In-Memory Database Systems**. In *Proceedings of 4th International Conference on Data Management Technologies and Applications*, DATA 2015, page 326âĂŞ331, Setubal, PRT, 2015. SCITEPRESS - Science and Technology Publications, Lda. Available from: `https://doi.org/10.5220/0005557303260331`. 7

[23] Till Kolditz, Dirk Habich, Dmitrii Kuvaiskii, Wolfgang Lehner, and Christof Fetzer. **Needles in the Haystack — Tackling Bit Flips in Lightweight Compressed Data**. In Markus Helfert, Andreas Holzinger, Orlando Belo, and Chiara Francalanci, editors, *Data Management Technologies and Applications*, pages 135–153, Cham, 2016. Springer International Publishing. 7

[24] Till Kolditz, Dirk Habich, Wolfgang Lehner, Matthias Werner, and Stefan T.J. de Bruijn. **AHEAD: Adaptable Data Hardening for On-the-Fly Hardware Error Detection During Database Query Processing**. In

Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, pages 1619–1634, New York, NY, USA, 2018. ACM. Available from: `http://doi.acm.org/10.1145/3183713.3183740`. 7, 18, 26, 28

[25] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. **Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design**. *SIGOPS Oper. Syst. Rev.*, **42**(2):265–276, March 2008. Available from: `http://doi.acm.org/10.1145/1353535.1346315`. 1, 4, 43

[26] Xin Li, Michael C Huang, Kai Shen, and Lingkun Chu. **A realistic evaluation of memory hardware errors and software system susceptibility**. In *Proc. USENIX Annual Technical Conference (ATCâĂŹ10)*, pages 75–88, 2010. 1, 4, 43

[27] Xin Li, Kai Shen, Michael C Huang, and Lingkun Chu. **A Memory Soft Error Measurement on Production Systems.** In *USENIX Annual Technical Conference*, pages 275–280, 2007. 4, 39

[28] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. **RIFLE: A general purpose pin-level fault injector**. In Klaus Echtle, Dieter Hammer, and David Powell, editors, *Dependable Computing — EDCC-1*, pages 197–216, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. 7

[29] T. C. May and M. H. Woods. **Alpha-particle-induced soft errors in dynamic memories**. *IEEE Transactions on Electron Devices*, **26**(1):2–9, Jan 1979. 4

[30] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. **Fault injection techniques and tools**. *Computer*, **30**(4):75–82, April 1997. 7, 43

[31] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D. D. Mannaru, A. Riska, and D. Milojicic. **Susceptibility of commodity systems and software to memory soft errors**. *IEEE Transactions on Computers*, **53**(12):1557–1568, Dec 2004. 1, 5, 43

[32] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. **Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field**. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 415–426, June 2015. 1, 4, 39, 43

[33] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. **Cycles, Cells and Platters: An Empirical Analysisof Hardware Failures on a Million Consumer PCs**. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 343–356, New York, NY, USA, 2011. ACM. Available from: `http://doi.acm.org/10.1145/1966445.1966477`. 1, 5, 7, 43

[34] E. Normand. **Single event upset at ground level**. *IEEE Transactions on Nuclear Science*, **43**(6):2742–2750, Dec 1996. 4

[35] Meikel Poess and Chris Floyd. **New TPC Benchmarks for Decision Support and Web Commerce**. *SIGMOD Rec.*, **29**(4):64âĂŞ71, December 2000. Available from: `https://doi.org/10.1145/369275.369291`. 20

[36] Mark Raasveldt and Hannes Mühleisen. **DuckDB: an Embeddable Analytical Database**. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984. ACM, 2019. 17

[37] Richard D Schlichting and Fred B Schneider. **Fail-stop processors: an approach to designing fault-tolerant computing systems**. *ACM Transactions on Computer Systems (TOCS)*, **1**(3):222–238, 1983. 8

[38] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. **DRAM Errors in the Wild: A Large-scale Field Study**. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM. Available from: `http://doi.acm.org/10.1145/1555349.1555372`. 1, 4, 43

[39] Seungjae Han, K. G. Shin, and H. A. Rosenberg. **DOCTOR: an integrated software fault injection environment for distributed real-time systems**. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*, pages 204–213, April 1995. 8

[40] T. Siddiqua, V. Sridharan, S. E. Raasch, N. DeBardeleben, K. B. Ferreira, S. Levy, E. Baseman, and Q. Guan. **Lifetime memory reliability data from the field**. In *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, Oct 2017. 1, 4, 5, 43

[41] Volkmar Sieh. **Fault-injector using UNIX ptrace interface**. In *Internal Report 11/93, IMMD3, Universität ErlangenNürnberg*. Citeseer, 1993. 10, 39

[42] D. Skarin, R. Barbosa, and J. Karlsson. **GOOFI-2: A tool for experimental dependability assessment**. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 557–562, June 2010. 8

[43] Vilas Sridharan, Jon Stearley, Nathan DeBardeleben, Sean Blanchard, and Sudhanva Gurumurthi. **Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults**. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 22:1–22:11, New York, NY, USA, 2013. ACM. Available from: `http://doi.acm.org/10.1145/2503210.2503257`. 1, 4, 5, 43

[44] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. K. Iyer. **NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors**. In *Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000*, pages 91–100, March 2000. 8

[45] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. **A survey on fault injection techniques**. *Int. Arab J. Inf. Technol.*, **1**(2):171–186, 2004. 8

[46] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. **IBM experiments in soft fails in computer electronics (1978âĂŞ1994)**. *IBM Journal of Research and Development*, **40**(1):3–18, Jan 1996. 4

[47] James Ziegler, M.E. Nelson, J.D. Shell, Roy Peterson, C.J. Gelderloos, H.P. Muhlfeld, and C.J. Montrose. **Cosmic ray soft error rates of 16-Mb DRAM memory chips**. *Solid-State Circuits, IEEE Journal of*, **33**:246 – 252, 03 1998. 4