**Fault Tolerance Methods and Implementations in Big Data Applications**

Nianfeng Tzeng, Austin Pohlmann, Purushottam Sigdel

**Abstract**

*For many years now, the fault tolerance mechanisms built into hardware have been thought of as adequate. One such example is Error-Correcting Code (ECC) memory. This form of memory (most frequently) uses Single Error Correction and Double Error Detection (SECDED) Hamming code. This proved to be a great solution years ago, but more modern systems are beginning to require more fault tolerance than this scheme is able to provide. While there are better methods of hardware fault tolerance, the more thorough methods usually incur chip area, performance, and power overheads. Even systems that initially have adequate tolerance mechanisms will eventually fall victim to other factors that may induce more errors than can be handled. Many older systems will fall victim to system aging; the original hardware will begin to underperform as it ages and wears. Even new systems might lose a small amount of stability under higher loads; higher loads cause more heat to be produced, which can have a negative effect on the running machine, or worse, permanently damage the hardware.*

1. **Introduction**

With hardware fault tolerance methods beginning to lose reliability in modern systems with modern needs, software fault tolerance has begun to gain more traction.

Software fault tolerance can come in numerous varieties: large overhead, small overhead, general protection, algorithm/implementation methods, high fault coverage, and low fault coverage.

In this paper, we evaluate the effects of various forms of fault tolerance on a handful of big data applications. The applications are from BigDataBench and have been modified to include various fault tolerance techniques as described in [1]. Alongside those integrated fault tolerance mechanisms, we used Berkley Labs Checkpoint-Restart to perform kernel-level checkpointing.

The applications from [1] used Kontrollable Utah LLVM Fault Injector (KULFI), a fault injector capable of doing static and ‘dynamic’ fault injection. However, the dynamic portion was the program deciding where to inject the fault when it is first run; once a program starts, the point of injection was fixed. This created a conflict: we were unable to restart from checkpoints without running into the exact same fault being injected again. As a result, we created a fault injector of our own to meet our needs.

1. **Implemenation**

Seeing that KULFI would be insufficient for our tests, we created a fault injector of our own. We created a linux kernel module in C for the heart of the injector.

The module, when loaded, creates an entry in /proc/ and implements the read and write functions for that entry. The write function is used for injection. When the user writes the string of the process ID (PID) to the proc entry (there is also optional functionality to specify which register and which bit to inject), the module will find the task\_struct (the data structure used to represent tasks in the linux kernel) and send a STOP signal to the process. The module will sleep for up roughly 1 millisecond while waiting for the process to enter the STOPPED state. In the event the sleep times out, the injection process is aborted. This is to prevent injection from having a significant impact on the runtime of the application being injected. If the task is successfully stopped, the injection proceeds. When the task is in the stopped state, the contents of the CPU registers at the time it was stopped is stored into the task\_struct of the process. From there, we can access and modify the contents of the registers. Two random numbers are generated to determine which register and which bit to flip. The information on which process, register, and bit flipped is stored for later use in the read. It should be noted that we do not inject bit flips into the Program Counter (PC) register. After the injection procedure, the process is sent the CONTINUE signal to resume running. When it runs, it will load the registers back from the task struct, which will include loading the modified register into the CPU. The read function will return a string containing the information about which process was last injected, which register was modified, and which bit was flipped. The string represents the details of the most recent injection.

This scheme of fault injection would allow us to inject errors on the fly, enabling us to use checkpointing properly. From that point we also created a tool to control fault injection while a program is being executed. There were a few other needs we needed to address: to inject faults at a random but fairly consistent interval, create checkpoints at regular intervals, and control the restoration from checkpoint in the event of abnormal program termination. The final application was written in Go.

To handle error injection, we needed a way to inject at random intervals, but still have control over the frequency. It was decided that randomly sampling from an exponential distribution would give us the results we desired. Our mediator program would then need to take in a time interval to be used as the lambda (average time between occurrences in the distribution).

The next issue that needed to be addressed by our mediator program was checkpointing. As it stands, BLCR is a command line utility that one can use to manually create a checkpoint and restart from one. We needed a way to checkpoint at fixed intervals and a scheme to restore from checkpoints. The scheme we ended up using involved 2 parameters: the number of checkpoints to keep (these can get pretty big sometimes, so we need to have a limit on the number we keep) and the number of times to retry from each checkpoint before assuming it was a bad checkpoint. The latter has slightly more complicated purpose. In the event an injected error does not cause an immediate issue, it could potentially be checkpointed. Any failure after said checkpoint might or might not be due to the previously injected error, so this parameter gives us some wiggle room. Only retrying once, we would discard the checkpoint if another failure occurred and try from an earlier one. Retrying 2 or 3 times from a single checkpoint before assuming it’s a bad one will help us be sure we aren’t deleting a good checkpoint in the event of unrelated errors causing failure.

The final part of our mediator program needs to track how many checkpoints were created, how many faults were injected, how much time was spent creating checkpoints, how much time was spent injecting errors, and how long the target program took to run.

Considering the two major functions of our program are checkpointing and injecting, we’ll be referring to it as auto checkpoint-inject, or autoci.

With our new fault injector and autoci complete, we were able to begin testing various configurations with five different applications: quicksort, set union, matrix multiplication, MD5, and metropolis hastings. We ran tests with and without LSU’s individual fault tolerance implementations, with and without checkpointing, with different amounts of backups to keep, and with different retry amounts for each checkpoint. We did 300 runs for each configuration and recorded the average runtime for successful runs (finished execution and produced correct results) what the percentage of runs that were successful out of all of the runs.

1. **Results**

The results varied greatly for each application. The implementation of each algorithm had different effects on the behavior of each application when injected with faults. Applications with large amounts of pointer arithmetic tend to experience more abnormal terminations and therefore benefit more from checkpointing than from most concurrent error detection (CED) methods. Application more prone to silent data corruption (SDC) will see less of a benefit from checkpointing. Figure 1 shows the average runtimes for each application, along with running it with BLCR and with CED. M. Hastings was an example of an application with a very low BLCR overhead. It uses very little memory and primarily does computation. The CED implementation for each program varies greatly, and each one provides a different amount type of fault coverage. While matrix multiplication has the smallest CED overhead and MD5 has the largest, each provides different amounts of fault coverage as seen in figured 2 and 3. Some of this fault coverage comes at a heavier cost. MD5 was an example of an application whose CED implementation caused significant overhead when in use. On the other hand, quicksort received a larger benefit with a minimal amount of overhead. For programs that heavily use pointers, such as matrix multiplication, BLCR provided a much larger benefit than CED with a very small overhead.

Afterwards, we tried running the same applications with slightly different settings for BLCR. The different were configurations are as follows: keeping the three most recent checkpoints and restarting from each one no more than three times (Figures 4 and 5), keeping the two most recent checkpoints and restarting from each one no more than two times (Figures 6 and 7), keeping the two most recent checkpoints and restarting from each one no more than one time (Figures 8 and 9), and keeping the most recent checkpoint and restarting from each one no more than one time (Figures 10 and 11). While most results were expected, there are a few odd occurrences to explain. In the union application in Figure 2, it appears that the CED experienced more failure than the version without it. We believe that this particular implementation of CED does not cover the effected areas thoroughly, and only introduces more areas for faults to occur. In a few other situations, the average time of successful runs was faster that the average time of runs without any fault tolerance and no faults injected.

Due to the nature of our fault injector, we were unable to replicate identical faults to test single fault coverage effectively. Since the register and bit is selected randomly, it is very possible that the target might be a register that is not currently live or a bit that won’t be accessed (injecting the upper 32 bits of a 64 bit register that holds a 32 bit value).

1. **Conclusion**

From these results, we have come to the conclusion that both forms of fault tolerance will have different overheads and fault coverage on a per application basis. A single application can even have multiple forms of CED, each able to address different issues and incurring different overheads. We observed that traditional CED methods do not typically do well with protecting pointers, and programs vulnerable to pointer corruptions would find a much greater benefit in checkpointing. Checkpointing, however, is unable to solve the issue of silent data corruption.

Great examples of CED that have been very finely tailored to the application are the many implementations of Fast Fourier Transformations (FFT). Most of these are able to use certain properties of the operation to be able to detect (and in some cases correct) faulty behavior. A particularly elegant CED method intended for databases can be found in [2].

1. **References and Figures**

[1] LeCompte, T., Legrand, W., Chen, S., Peng, L. “Soft error resilience of Big Data kernels through algorithmic approaches”, J Supercomput (2017). Doi:10.1007/s11227-017-2042-6

[2] Till Kolditz, Dirk Habich, Wolfgang Lehner, Matthias Werner, and Stefan T.J. de Bruijn. 2018. 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). ACM, New York, NY, USA, 1619-1634. DOI: https://doi.org/10.1145/3183713.3183740

**Figures listed in order**

