J Supercomput
DOI 10.1007/s11227-017-2042-6

Soft error resilience of Big Data kernels through algorithmic approaches

Travis LeCompte[1] · **Walker Legrand**[1] · **Sui Chen**[1] · **Lu Peng**[1]

Abstract As the volume of data generated each day continues to increase, more and more interest is put into Big Data algorithms and the insight they provide.? Since these analyses require a substantial amount of resources, including physical machines, power, and time, reliable execution of the algorithms becomes critical. This paper analyzes the error resilience of a select group of popular Big Data algorithms and shows how they can effectively be made more fault-tolerant. Using KULFI (http://github.com/quadpixels/kulfi, 2013) and the LLVM (Proceedings of the 2004 international symposium on code generation and optimization (CGO 2004), San Jose, CA, USA, 2004) compiler for compilation allows injection of artificial soft faults throughout these algorithms, giving a thorough analysis of how faults in different locations can affect the outcome of the program. This information is then used to help guide incorporating fault tolerance mechanisms into the program, making them as impervious as possible to soft faults.

Keywords Soft faults · Fault injection · Numerical errors · Fault resilience · Big Data kernels

1 Introduction

In today's world, Big Data processing has become progressively more prevalent. A large percentage of the world's population spends hours every day connected to the Internet in some way. This continuous usage by such a large population generates

✉ Lu Peng
lpeng@lsu.edu

[1] Division of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, Louisiana, USA

Published online: 18 April 2017

an immense volume of data to process, approximately 2.5 exabytes of data per day as reported by IBM in 2012 [2]. One can imagine that this number will only grow in the future. To handle this processing, most companies utilize high-performance computers such as supercomputers or computing clusters. As these high-performance computers become more and more advanced, one primary focus is to minimize the amount of power required by a processor to perform operations. This is combined with the shrinking feature size of processor circuitry which is expected to reach 5–7 nm, which is just 10–14 silicon atoms across, in 2020 [24]. Traditionally, error resilience is focused on natural events that can affect the contents within computer clusters [8], such as charged alpha particles or cosmic rays. With new circuitry technologies and low-power operations for energy savings, errors can arise form more varied usage conditions (such as high temperature/altitude zones/vehicles), 3D interconnect and chip structures, etc. [24].

A recent study reveals the mean time between failure (MTBF) of double bit errors in the Tesla K20 GPUs used in the Titan supercomputer [30] is as low as 160 h (about one error per week.) This is 3 magnitudes smaller than the manufacturer-rated MTBF of 219,282 h under a controlled environment [28]. As such, Big Data practitioners who seek to build clusters using commodity hardware (which may not have ECC like the K20 does) may not be able to consider soft faults to be an impossibility. The result is errors on less protected systems may go unnoticed. Thus, it has become increasingly important for programs to detect and prevent these faults on their own.

There exists a large range of Big Data algorithms for many specific applications, ranging over regression and classification to simple statistical reporting. One cannot hope to examine each program individually to study its fault-tolerant potential. However, it is common for programs to share features and reuse basic algorithms. Researchers have come up with proposals with the goal of characterizing Big Data programs with simpler benchmarks, including HiBench [22], BigBench [20], AMP Benchmarks [5], YCSB [15], LinkBench [7], CloudSuite [17], and BigDataBench [19]. The latest one of the collection, BigDataBench, identifies eight Big Data kernels or dwarves that are used by a significant number of these programs: linear algebra, sampling, transform operations, graph operations, logic operations, set operations, sort, and statistic operations. Thus, we believe that studying the fault tolerance of one representative from each of these kernels will provide insight into potential fault-tolerant mechanisms for Big Data overall. In this paper, we selected the following eight algorithms to represent each data dwarf, respectively: matrix multiplication, Markov chain Monte Carlo, fast Fourier transform, breadth-first search, MD5, union set operation, quicksort, and GREP. We observe the fault-tolerant potential of each algorithm by identifying algorithm-specific invariants that, when violated, indicate the occurrence of a soft fault. These invariant checkers are implemented into each algorithm along with a recovery system. Faults are then injected during the execution of the algorithm with fault injection tools such as KULFI, and the resulting behavior is observed. We show that these fault-tolerant systems reduce the impact of these faults by lowering both incorrect answers and execution failures. This provides information into the effectiveness of this method of fault tolerance on Big Data applications in general, and the value of fault resilience in Big Data algorithms. Our experiments demonstrate that the soft error resilience will be significantly improved with the proposed methods.

2 Related works

The study of error resilience and related fields such as uncertainty quantification has been mostly focused on scientific computing so far [10]. Error resilience is a must for highly unreliable environments such as on an unmanned aerial vehicles [31], especially with the increasing processing power of onboard computers.

Numerical errors are more complicated than their integer counterparts as floating-point operations are not exact and dependent on order of operation. This can be seen in parallel reduction [14] and the linear algebra routines in BLAS [23]. Arithmetic-heavy applications including physics-based simulations are designed to cope with round-off errors [32]. With finite bit precision, floating-point operations and their results can be seen as approximations. Recently, NVIDIA began to provide half-precision (FP16) floating-point arithmetics [6] with the aim to boost performance at negligible cost of accuracy, particularly in deep-learning applications which are closely related to big data applications. From this point of view, approximation and tolerance to soft faults are very similar in nature.

The fault resilience workflow encompassing fault injection and resilience can be done on multiple levels of the hardware–software stack. Existing works have utilized actual proton sources on the physical level [9], embedded hardware sensors on the circuit level [27], FPGAs on the digital logic level [29], full-system simulators and virtual machines on the architectural level [21], and debugging utilities on the high application level [31].

Fault resilience study can be costly on the experiments side as well as on the engineering side. Several works have proposed remedies: to reduce the huge size of the fault injection experiment space, Relyzer [25] exploits control flow and value equality to prune the fault injection space; to save engineering cost on large code bases, a programming model called containment domains [13] provides developers a hierarchical view of fault resilience. Modular analysis [12] provides a first step toward lowering the cost in fault injection and the understanding of numerical error propagation. We expect to see fault resilience get increased support in the future in tool chain and modeling just like profiling and debugging do.

In this paper, we utilized a lightweight fault injector KULFI [4], which is built on LLVM [26] and extended in a previous work [11] to analyze the fault tolerance of Big Data algorithms [12]. However, we further extend it to constructing invariant checkers and recovery systems. These let us observe program behavior with injected faults at any location in program execution.

To keep the cost of fault injection experiments at an acceptable level, we use a statistical tool to ensure our experiments are statistically grounded and can represent the fault injection space with a sample that is much smaller than the experiment space size.

3 Design procedure

For each of the big data dwarves included, we follow a general approach shown in Fig. 1 to analyze the fault tolerance of the dwarf. This process involves identifying a specific

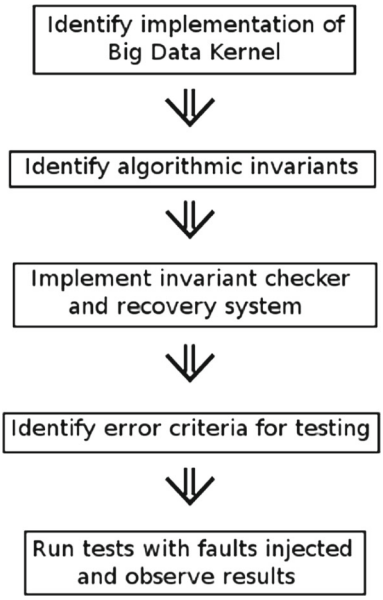

Fig. 1 Method of testing and improving fault tolerance of selected programs

implementation of an algorithm to test and represent the dwarf, which must in turn be compatible with KULFI and LLVM; identifying one or more invariants within the algorithm; implementing the said invariant(s) to check for errors, along with recovery in the event any invariant is violated; identifying an error criteria, to allow for detecting improper program output; and lastly, injecting faults into program execution during tests to observe the effects of the invariant implementation and recovery system.

To identify implementations for testing, we searched for published implementations of algorithms that we consider exemplified the dwarf in question. This search typically began with the BigDataBench benchmark suite itself, though some compatible implementations were difficult to find and are taken from public github repositories.

Next we attempt to identify invariants within the algorithm for use in identifying errors during program execution. Some implementations are relatively simple, such as grep, and do not exhibit high-level invariants. For these implementations, we choose to use redundancy in critical operations to eliminate errors. For those that do contain invariants, we then implement the check for the invariant along with a recovery system. Thus, if faults are injected into the program, the invariants potentially fail and the program recovers from the fault, instead of allowing the fault to propagate to output error or program failure.

However, we must be able to detect whether a program is creating proper output or not. These criteria are algorithm specific, typically involving a comparison of outputs or of statistics for the outputs. For some programs such as union, this is very straightforward, while others such as breadth-first search or Markov chain Monte Carlo (MCMC), which relies on random sampling, are more complex.

Lastly, we need to test the effectiveness of the fault-tolerant additions. This testing includes a minimum of 5000 trials for both fault-tolerant and non-fault-tolerant

algorithms each, along with varying-sized data sets for some algorithms. Faults are injected dynamically into program execution using KULFI, and outputs for both the fault-tolerant and non-fault-tolerant versions are compared with the "correct" program output, as determined by a non-fault-tolerant execution with no injected faults. This comparison gives a metric to determine whether a trial is incorrect, and how incorrect it is.

4 The eight dwarves

4.1 Linear algebra (Matrix Multiplication)

4.1.1 Introduction

Matrix multiplication is a common linear algebra operation that involves many individual multiplications and summations. For this purpose, we use the GSL (GNU Scientific Library) [18] for matrix and vector multiplication. Specifically, we multiply two matrices A and B together four different times, once for each combination of transposition (i.e., transposed A, B, A and B, or neither). This is because GSL provides four paths for each combination, each with an optimized access pattern that tries to maximize cache efficiency for the particular combination. With the four combinations, we can fully evaluate the resilience of GSL's implementation of this operation.

4.1.2 Invariants

Suppose that we have matrices A, B, and C such that $AB = C$. Then for any vector v, $ABv = Cv$. We can then check the results of multiplication by using a nonzero vector v. While this is obvious, it is useful due to the associative property of matrix multiplication. The multiplication can be associated as $A(Bv) = Cv$, which in turn avoids any direct matrix–matrix multiplication, relying only on matrix–vector multiplication, which is substantially less expensive to compute.

4.1.3 Testing procedure

As previously stated, the algorithm multiplies matrices in four different transposition combinations to fully test the multiplication algorithm. For each one of these four multiplications, the correct output matrix as calculated by the non-fault-tolerant algorithm with no faults injected is stored to a file. These precomputed matrices are then read in by the algorithm during testing, where it compares the results of the current trial to the precomputed matrices. A root-mean-square difference is calculated by comparing each element of the current trial's matrix to the corresponding element in the precomputed matrix. This comparison includes a tolerance of $1e{-}8$ to account for potential rounding errors. Thus, an output is generated including one root-mean- square difference for each one of the four multiplications. Both the non-fault-tolerant and the fault-tolerant versions are run on 10,000 trials each, using matrices of size 500×400 and 400×300.

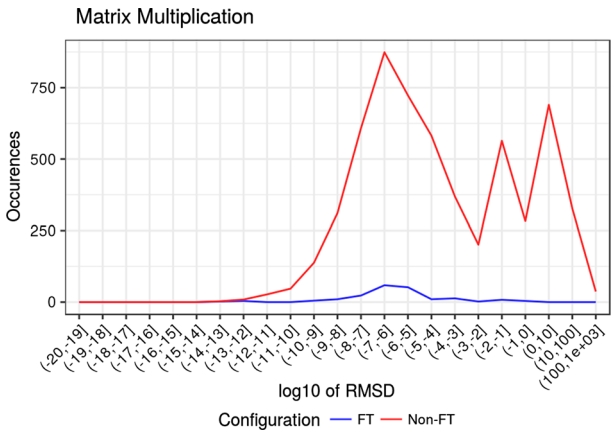

Fig. 2 Number of incorrect results by magnitude of error in incorrect outcomes

4.1.4 Results

For the purpose of consolidating data, the individual error of each of the four multiplications performed in the algorithm has been summed into one total error. For 10,000 trials, the non-fault-tolerant version of the algorithm produced 5793 wrong answers and 3501 segmentation faults (program execution failures), while the fault-tolerant version produced 192 wrong answers and 3,503 segmentation faults. This is a massive reduction in wrong answers by 96.68% and a small increase in segmentation faults by 0.05%. Additionally, the non-fault-tolerant version has a very large mean error of $7.07e{+}298$, while the fault-tolerant version has a mean error of only $4.57e{-}03$. Thus, only the smallest errors still occur in the fault-tolerant version, while massive errors occur in the non-fault-tolerant version.

The included histogram, Fig. 2, shows the number of occurrences of error that occur, grouped by the base ten logarithm of the consolidated error. First to note is the large reduction in small errors that occur. Second is the complete elimination of larger errors, even though they are not extremely common in the non-fault-tolerant version to begin with. However, since the errors are extremely large, they have a large influence on the mean error.

Error probability Next we analyze the probability of errors occurring based on the dynamic fault site the fault is injected into in Fig. 3. Dynamic fault sites in KULFI essentially count the instructions executed as they are encountered and label them with their position, which is referred to as the dynamic fault site ID (DFSID). Thus, analyzing the program behavior by DFSID is similar to examining the probability of error occurring at certain points of the program's execution and is determined by the number of injections at a particular dynamic fault site that cause an incorrect answer. The dynamic fault sites in turn map to a specific line of source code, so this effectively measures the probability of a wrong answer occurring due to an error in a specific line of source code. Due to a difference in dynamic fault site count, in the following graphs the non-fault-tolerant data points have been stretched to match

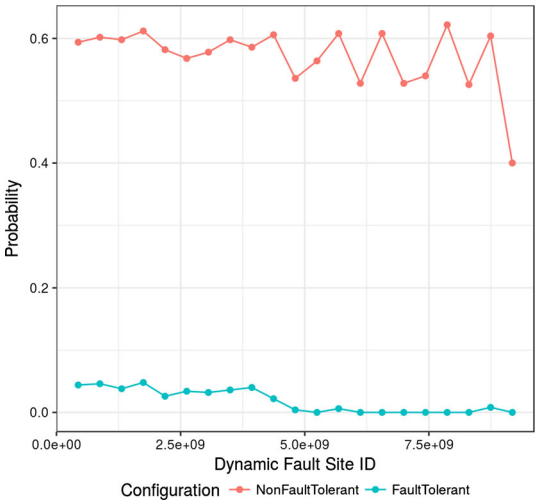

Fig. 3 Probability of error in non-fault-tolerant versus fault-tolerant runs by location of injected bit flip in Matrix Multiplication

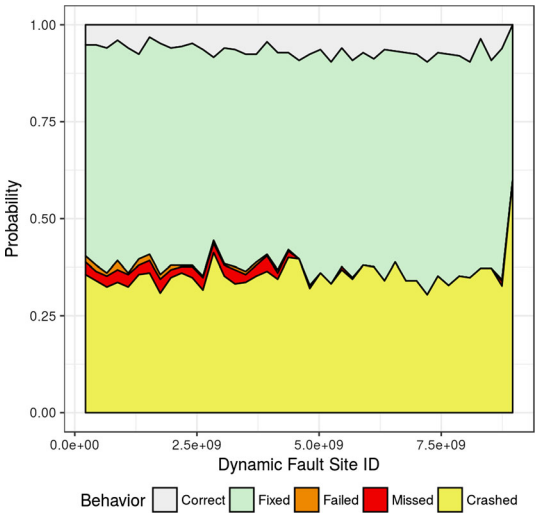

Fig. 4 Probability of the fault chcker in Matrix Multiplication crashed, fixed an error, failed to fix an error, failed to detect an error, or resulted correctly without fault check assistance

the fault-tolerant data points. Thus, both curves represent a full run of the tested program.

It is clear that there is a massive reduction in error probability across the board, nearly removing all error. The probability of error is relatively constant at the beginning of the program in both versions, but the fault-tolerant version removes the majority of error in the second half of execution, while the non-fault-tolerant version becomes more erratic. Overall, the fault-tolerant version has much more stable behavior, in addition to a large reduction in the probability of error consistent with previous results.

Behavior distribution Lastly, in Fig. 4, we analyzed the behavior of the fault-tolerant algorithm by dynamic fault site ID. The outcomes are classified as one of the following five: correct (no error detected, correct answer), fixed (error detected, correct

answer), missed (no error detected, incorrect answer), failed (error detected, incorrect answer), and crashed (program failed to complete execution). This gives a finer breakdown of how the fault tolerance works, by comparison to simply a right or wrong answer.

It becomes immediately clear that there are a large number of fixed outcomes, demonstrating that the added fault tolerance code has a large effect on program behavior, in accordance with previous results. Additionally, there are a large number of segmentation faults, as mentioned earlier. There are only a small number of correct outcomes, meaning that the faults have a large effect on the behavior of the program, and a small number of missed and failed outcomes, signaling a minor failure of the fault-tolerant additions.

4.2 Sampling (MCMC, namely Metropolis–Hastings)

4.2.1 Introduction

Markov chain Monte Carlo (MCMC) generally is a popular probabilistic algorithm that approximates an unknown distribution by sampling from said distribution and adjusting a test distribution based on samples. In this sense it acts similarly to a discrete Markov chain, the test distribution is adjusted to slowly approach the unknown distribution, and the transitions made rely only on the current state of the test distribution and the samples from the unknown distribution. When a large number of transitions are made, the test distribution comes to be approximately equal to the unknown distribution.

The specific MCMC implementation used in the following tests is a Metropolis–Hastings algorithm [1]. Metropolis–Hastings is commonly used to generate many random samples from distributions that would otherwise be difficult to sample. The algorithm first calculates an acceptance value $A(x)$ for a proposed sample x. Then if $A(x)$ is greater than an acceptance criteria probability $P(x)$, the proposed sample is accepted and the Markov chain changes state. This implementation relies heavily on the C++ Boost library for random number generation and probability distributions.

4.2.2 Invariants

The invariants used this algorithm are primarily related to the transitioning of the Markov chain, as the samples fed to the algorithm are random and thus cannot be bounded well.

For the acceptance criteria probability $P(x)$, $0 \leq P(x) \leq 1$. This holds true since $P(x)$ is indeed a probability, and thus should be bound by zero and one. Note that this does not extend to $A(x)$, where an acceptance value greater than one is allowed, which would indicate guaranteed acceptance.

For a transition in the Markov chain from state X to state Y, after the transition is complete, the current state of the chain should be equal to state Y. This is clear by definition of the transition, but should be checked to guarantee proper transitions.

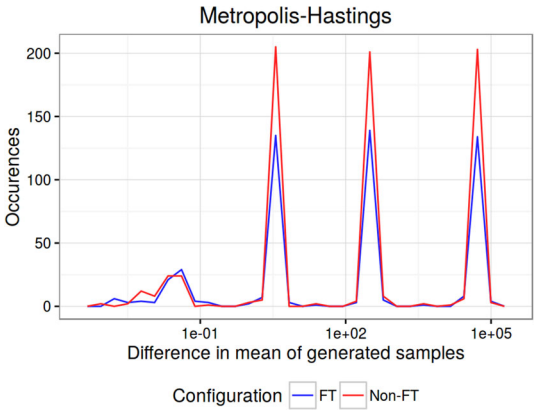

Fig. 5 Number of incorrect results by magnitude of error in incorrect outcomes

4.2.3 Testing procedure

Both the fault-tolerant and non-fault-tolerant versions of the algorithm generate 10,000 samples from a distribution with a burnin of 500. This leads to a total of 5,000,000 potential samples generated and tested, and the same number of potential transitions, per execution. Each version of the algorithm was run for a total of 10,000 trials. The correctness of the algorithm was determined by the absolute value of the difference between the mean of all 10,000 samples generated in the trial in question and the mean of all 10,000 samples generated by the non-fault-tolerant version with no faults injected during execution.

4.2.4 Results

Error occurrences For the 10,000 trials run, the non-fault-tolerant version of the algorithm produced 646 wrong means and 473 segmentation faults, while the fault-tolerant version produced 442 wrong means and 429 segmentation faults. This shows an improvement of 31.58% in errors and 9.31% in segmentation faults. This merely minor reduction in segmentation faults is due to the fact that the majority of segmentation faults occur in the sources for the Boost library. Interestingly enough, the fault-tolerant version has a mean difference of 17,744.94, while the non-fault-tolerant version has a mean difference of 17,740.21. So while there is an overall reduction in both the number of wrong outputs and failures in execution, the erroneous outputs are actually larger in difference on average, though not by much.

It is interesting to note in Fig. 5 that there are only three major peaks of error, most likely caused by a pattern in the fault injection. It is clear that the fault-tolerant version reduces these major peaks of error substantially. However on the less prominent peaks, occasionally the fault-tolerant version overtakes the non-fault-tolerant version in number of occurrences of error.

Error probability The graph in Fig. 6 oscillates frequently, and the fault-tolerant version occasionally overtakes the non-fault-tolerant version, similar to the previous graph in

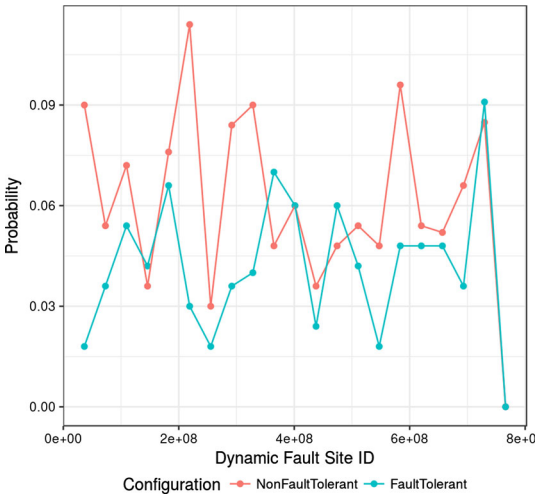

Fig. 6 Probability of error in non-fault-tolerant versus fault-tolerant runs by location of injected bit flip in MCMC

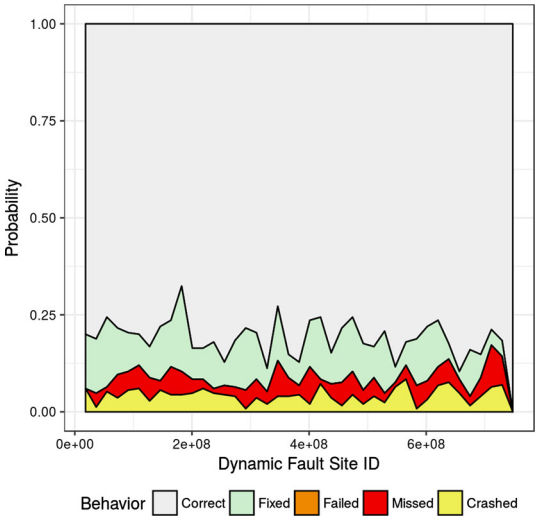

Fig. 7 Probability of the different outcomes of the fault checker in MCMC

Fig. 4. However, the fault-tolerant version cuts out the major peaks of errors and only slightly exceeds the non-fault-tolerant version when it does. The magnitudes of the probabilities are also very small, meaning that one error during execution greatly influences this graph.

Behavior distribution The first feature to note in Fig. 7 is the overwhelming percentage of correct outcomes by comparison to all other outcomes. This means that many of the faults injected during execution do not affect the outcome of the program. Next is the relatively small percentage of segmentation faults, which largely occur in the distributions of the Boost library. What remains are mostly fixed outcomes, with a smaller percentage of missed outcomes and no failed outcomes. This means that the

error correction is successful when the errors are detected, but there are errors that are not seen by the fault-tolerant code.

4.3 Transform operations (FFT)

4.3.1 Introduction

The fast Fourier transform algorithm was developed to quickly compute the discrete Fourier transform, and its inverse, of a data set of any size. The discrete Fourier transform is used on complex waveforms and is widely applicable to many fields of science.

4.3.2 Fault tolerance analysis

The best way to determine whether the FFT algorithm was properly executed is to perform a check after the transform is performed. The best check for this algorithm is a theorem called Parseval's theorem [3] which goes as follows:

$$\sum_{n=0}^{N-1} |x_n|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X_k|^2$$

The small x indicates the data signal before transformation, and the capitalized X represents the transformed data signal. Computing these summations on both data sets after the algorithm has been run and comparing them with a tolerance of 0.001 to account for rounding errors tells whether or not the transform has been returned properly and whether re-computation is necessary.

4.3.3 Testing procedure

The algorithm was run on an input signal containing 524,288 complex values. Faults were injected over the course of around 20,000 runs of the algorithm, both with and without fault checks.

4.3.4 Results

The fault check proved to be quite effective correcting wrong answers. Out of 20,000 fault-injected runs on the original algorithm, 14,119 came out as wrong answers. With the fault checks implemented, only 972 out of 20,000 runs returned as faulty.

Error probability The graph in Fig. 8 was generated to show the probability of error in the fault-injected runs with and without the implemented fault checks. As shown in the graphs, over 90% of the fault- injected runs without fault checks turned up as incorrect answers. This percentage drops to roughly 5% with the addition of the fault checks.

Behavior distribution The graph in Fig. 9 shows that a large portion of the runs of the program ended up being detected as faulty and were corrected by the fault checks. The

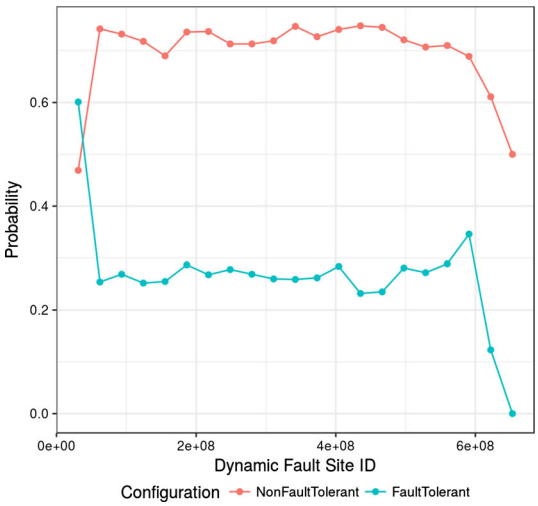

Fig. 8 Probability of error in non-fault-tolerant versus fault-tolerant runs by location of injected bit flip in FFT

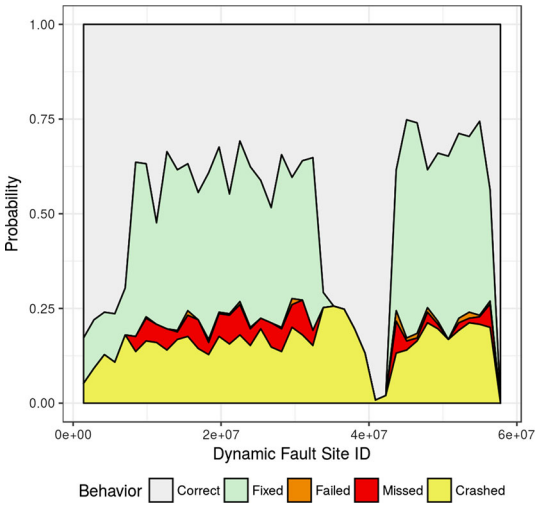

Fig. 9 Probability of the different outcomes of the fault checker in FFT

program also ended up crashing roughly 25% of the time most likely due to the fact that the complex numbers were contained in arrays which can return segmentation faults if a fault occurs in an array offset.

4.4 Graph operations (BFS)

4.4.1 Introduction

Breadth-first search (BFS) is a common graph search algorithm that searches for an element from a root node level by level, such as reading a book (thus the name). This specific implementation constructs a search tree with a given root node from a given

edge list. This effectively finds the shortest path from the root node to each other node. The algorithm itself heavily uses C arrays for operation.

4.4.2 Invariants

Most of the fault-tolerant additions to the BFS algorithm are duplications to critical assignments or if statements to guarantee proper execution. Due to the reliance on arrays, the most common errors observed are segmentation faults due to incorrect array pointer offsets. Second to this are errors in loop conditions or control variable assignments and updates. Therefore, most additions of fault-tolerant code involve assuring proper variable assignment or control flow conditions at critical points of program execution. These together help ensure the following invariants in the tree are not violated.

Let `n` be the number of nodes specified by the input file. After the breadth-first traversal, the produced tree should contain the same `n` nodes. The algorithm should not remove any nodes from the graph, and it should only traverse the edges to find shortest paths in a breadth-first manner.

Let `depth[v]` be the depth of a node `v`, and let `parent[v]` be the parent node of a node `v`. Then after the breadth-first traversal, `depth[v] > depth[parent[v]]` for any node `v` that is not the root node (which has no parent). This holds because BFS searches in descent from the root node level by level and ignores cycles in the graph, preventing two nodes at the same depth level from having a parent–child relationship. Therefore, the only parent–child relationships that exist are those in which the depth of the parent is less than the depth of the child.

Let `distance[v]` be the distance from node `v` to the root of the tree. Then, after the traversal, `distance[v]` should be the length of the shortest path from node `v` to the root node. This is a common invariant of BFS which has been proven in classroom lectures [16].

4.4.3 Testing procedure

To test the BFS algorithm under fault injection, we ran the algorithm on both small (1,024 nodes, 10,240 edges) and large (16,384 nodes, 262,144 edges) data sets. In both cases, both the fault-tolerant and non-fault-tolerant versions of the algorithm were run for a total of 10,000 outcomes using the same root node from which to traverse the tree. Faults were injected into both the fault-tolerant and non-fault-tolerant code, ranging over the full length of possible fault sites. These faults could be injected into the bit positions 1, 8, 15, 22, or 29. Error in a given outcome was measured by comparing the final tree structure output by the execution in question to the same output from a non-fault-tolerant, non-fault-injected BFS run on the same input data and counting the number of nodes which had a different parent, or path to the root node.

4.4.4 Results

Error occurrences Out of the 10,000 outcomes for each version of the algorithm on the small input graph, the non-fault-tolerant version has 1001 wrong outcomes

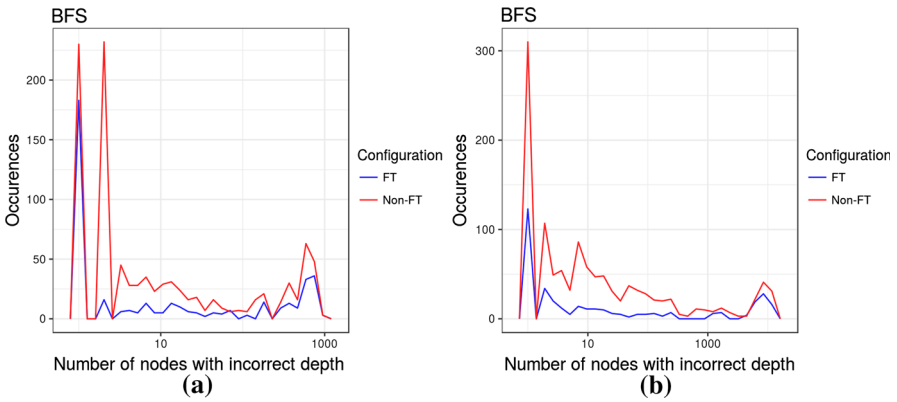

Fig. 10 Number of incorrect results by magnitude of error in incorrect outcomes in BFS. **a** Small input. **b** Large input

and 2427 segmentation faults, while the fault-tolerant version has only 412 wrong outcomes and 1709 segmentation faults. This shows an overall decrease in wrong outcomes by approximately 58.9% and a reduction in segmentation faults by 29.6%. The fault-tolerant version also has an average number of wrong values of 8.04 versus the non-fault-tolerant version's average of 14.9.

Out of the 10,000 outcomes for each version of the algorithm on the large input graph, the non-fault-tolerant version has 1258 wrong outcomes and 2241 segmentation faults, while the fault-tolerant version has only 359 wrong outcomes and 1596 segmentation faults. This shows an overall decrease in wrong outcomes by approximately 69.0% and a reduction in segmentation faults by 28.8%. The fault-tolerant version also has an average number of wrong values of 68.5 versus the non-fault-tolerant version's average of 117.1.

The results from the 10,000 outcomes from each version of the algorithm were compiled into a histogram in Fig. 10, counting the number of outcomes grouped together by their number of errors, which is scaled logarithmically.

The fault-tolerant version improves over the non-fault-tolerant version in both the small and large input cases, though by more so with large input. The greatest improvement comes in a large reduction of small errors, though there is improvement at nearly all occurrences of error. The error values are also not well concentrated outside of the very small areas of error, meaning that faults injected can cause a wide range of different output errors. Overall, it would appear that the algorithm performs better on the large input files, though they naturally have a higher average number of errors.

Error probability As shown in Fig. 11, there is clearly reduction in the probability of error at most locations in program execution, in both large and small input cases. There are certain places in execution, namely the start of the final quarter of execution, at which the fault-tolerant algorithm increases the probability of error. The program displays an interesting area of zero probability of error in the middle of program execution, in both the fault-tolerant and non-fault-tolerant versions. It is also worth

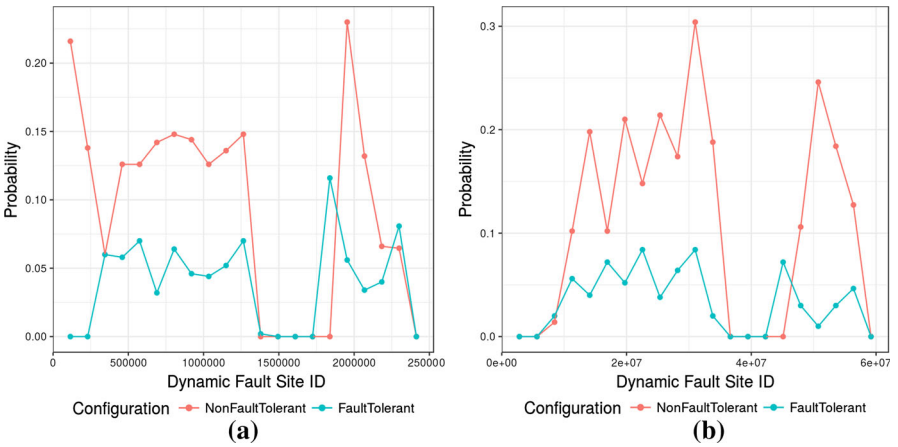

Fig. 11 Probability of error occurrences in BFS by dynamic site ID. **a** Small input. **b** Large input

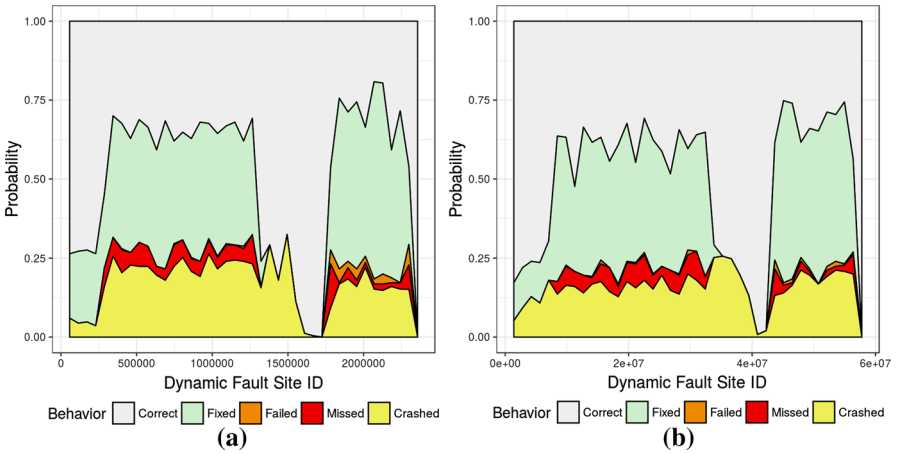

Fig. 12 Probability of the different outcomes of the fault checker in BFS. **a** Small input. **b** Large input

noting that the probabilities of error seem to have larger peaks in the large input file than the small input file.

Behavior distribution As shown in Fig. 12, the program displays a large number of crashed runs (segmentation faults) by comparison to some of the other categories. This is expected, as the program uses a large number of array operations. Secondly, the program is relatively fault-tolerant and many of the trials are already correct and do not detect any error or wrong output whatsoever. There is still a large portion of trials that are fixed, however, signaling the effect of the added fault-tolerant code. Lastly, there are a small number of failed/missed trials, and even less so in the large input trials. Overall, this means that the fault-tolerant code mostly either fixes errors, experiences no errors at all, or fails to complete execution.

```
MD5::uint4 z;
z = a;
if (z != a)
    z = a;
a = rotate_left(a + F(b,c,d) + x + ac, s) + b;
if (a != rotate_left(z + F(b,c,d) + x + ac + s) + b)
    a = rotate_left(z + F(b,c,d) + x + ac + s) + b;
```

Fig. 13 Code from the FF() function within the MD5 algorithm using fault checking techniques

4.5 Logic operations (MD5)

4.5.1 Introduction

MD5 was originally developed as a cryptographic hash function, the main purpose of which is to protect against intentional attacks on data that are being transmitted. In recent years, MD5 has been shown to be outdated and weak in protecting against intentional attacks, but it is still used today for checksum purposes and to verify data integrity.

4.5.2 Fault tolerance analysis

In the MD5 algorithm, there are four functions that are called many times each throughout the execution of the program. These functions, called FF, GG, HH, and II, respectively, are used to set up and call a function called `rotate_left` which performs the bitwise function of the same name on the data given by the function. In the original code for the algorithm, each of these four functions contains only one line:

- *FF*: `a = rotate_left(a + F(b,c,d) + x + ac, s) + b;`
- *GG*: `a = rotate_left(a + G(b,c,d) + x + ac, s) + b;`
- *HH*: `a = rotate_left(a + H(b,c,d) + x + ac, s) + b;`
- *II*: `a = rotate_left(a + I(b,c,d) + x + ac, s) + b;`

The same strategy was used to make each of the FF, GG, HH, and II functions fault-tolerant. Each function, which previously only contained a single line, was replaced with the code shown in Fig. 13.

A copy of the data contained in a is placed in the variable `z`. The data in `z` are then checked against a to make sure it was copied properly. The `rotate_left()` function is then called twice, once with the a variable and once with the `z` variable. The answers are compared, and if for some reason they are not equal, the answer is computed a third time and this answer is used as the output for the function. Due to the fact that this function is called numerous times throughout the program, this results in an instruction count overhead of 70%. Although there is a large addition of instructions when the fault tolerance checks are added to the program, the positive results of the fault tolerance appear to outweigh the drawbacks.

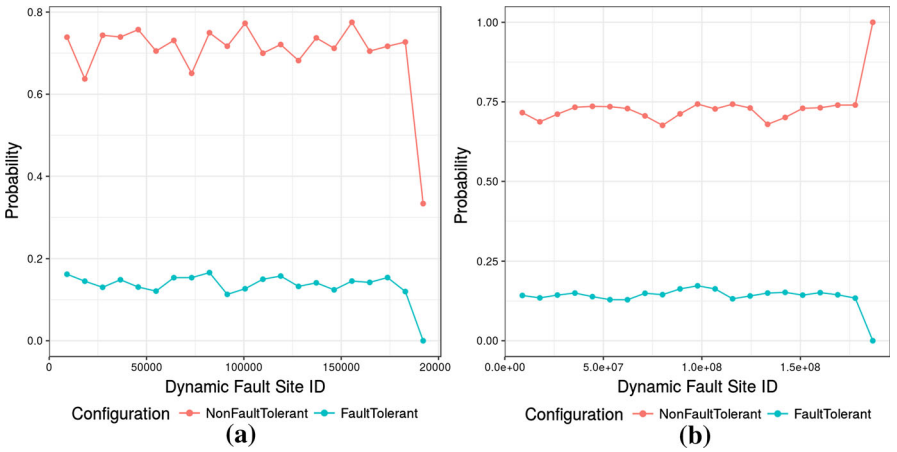

Fig. 14 Probability of error in non-fault-tolerant versus fault-tolerant runs of MD5 by location of injected bit flip. **a** Small input. **b** Large input

4.5.3 Testing procedure

To test the efficiency of the fault tolerance checks, the program was run thousands of times, injecting a single fault into a different location each run. To assure that the effectiveness of the fault tolerance checks was consistent regardless of input size, these fault injection trials were run on both a small and large input size. The small size was an input of 1000 randomly generated characters, and the large size was an input of 1,000,000 randomly generated characters. The program was run 50,000 times on each data set, both with and without the fault tolerance checks for a comparison of how often the program generated incorrect data.

4.5.4 Results

Line graphs in Fig. 14 were generated for both the small and large data sets in order to give a more visual representation of the probability of error. As shown in the graphs, it is clear that the fault tolerance checks create a large difference in error probability throughout the entire program (Fig. 15).

Error probability Line graphs in Fig. 14 were generated for both the small and large data sets in order to give a more visual representation of the probability of error. As shown in the graphs, it is clear that the fault tolerance checks create a large difference in error probability throughout the entire program.

Behavior distribution Both of the histograms in Fig. 15, which represent the small and large input trials, seem to be fairly similar and do not exhibit any major differences. In both graphs, roughly 25% of the trials crashed and did not run to completion due to some sort of error. Roughly 45% of the runs returned as "fixed" which shows that 45% of the runs were likely to return an incorrect answer but were corrected by the fault tolerance checks. The graph also shows that 20% of the runs resulted in correct outputs

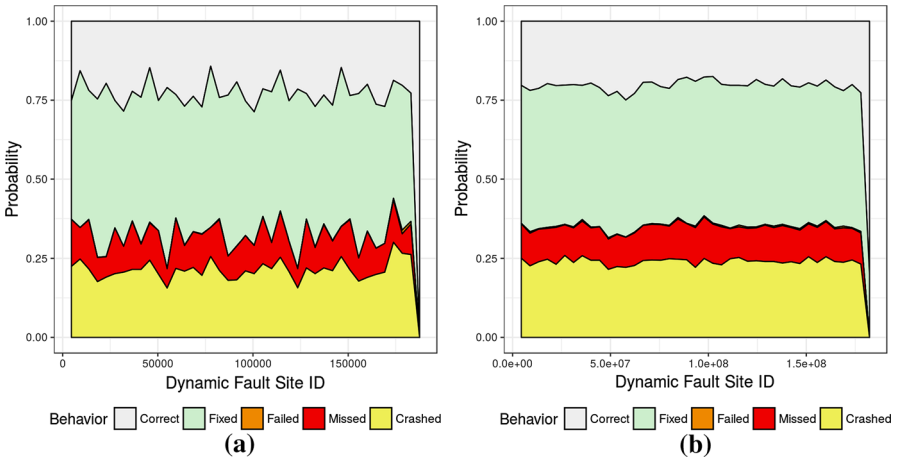

Fig. 15 Probability of different outcomes of the fault checker in MD5. **a** Small input. **b** Large input

(i.e., these runs were not affected by fault injection) and 10% resulted with incorrect outputs that were not detected by the fault checks. An extremely small (<1%) of the trials returned as "failed," indicating that an error was detected but was not fixed by the program.

4.6 Set operations (union)

4.6.1 Introduction

Union is among the most commonly used set operations and is applicable in any branch of math that deals with sets. A union functions purpose is to take two separate sets of data and output a single set which contains every element that occurs in at least one of the two sets. Also, the output of a union function should not contain the same element twice.

4.6.2 Fault tolerance analysis

The majority of the algorithm is just conditional statements and assignments of variables, so the best way to make the program resistant to faults that could potentially give a wrong output is to double check that these conditionals and assignments were properly executed without error. To do this, three different types of error checks were set up throughout the program. Though each of the checks only corrected a small number of wrong outcomes, together, they eliminate a sizable amount of wrong answers.

Assignments The first check that was used in the program, shown in Fig. 16a, was to check the assignment of elements contained in arrays. One incorrect bit could potentially lead to a snowball effect which could cause the output data to be incorrect

```
set3[count] = set1[cnt1];
if (set3[count] != set1[cnt1])
    set3[count] = set1[cnt1];
```

(a)

```
int tmp = cnt1 + 1;
cnt1++;
if (tmp != cnt1)
    cnt1 = tmp;
```

(b)

```
label2:
    while (set2[cnt2] == temp){
        ...
        ...
    }...
    if (set2[cnt2] == temp){...
        goto label2;
```

(c)

Fig. 16 Code examples from three operations in set union. **a** Assignment, **b** increment, **c** while loops

or even just make the program crash. This type of check was used several times for the assignment statements throughout the program.

Increment The second type of error check that was used in the algorithm, shown in Fig. 16b, was used to check to correctness of the multiple increment functions throughout the program. The variables that are incremented by one throughout the program are used to keep track of how many elements are being handled by union function. An error in one of the variable has the potential to drastically change the output. This check creates a copy of the variable to be incremented that is also incremented. The two are then compared and corrected if needed.

While loops The last type of check that was used to make the algorithm more fault tolerant, shown in Fig. 16c, was a check for while loops. Once the while loop is terminated, the program immediately double checks the conditional statement to assure that the while loop was terminated at the right time. If not, it goes back and continues with the while loop until it is supposed to be completed.

4.6.3 Testing procedure

Due to the frequent error checks that are necessary to keep the algorithm going in the case that a fault occurs, the fault-tolerant version contains roughly 70% more overhead instructions than the standard version. In order to study the effects that the fault tolerance checks had on the outcome of the program when faults are injected, we ran several trials of runs on the program. The first two trials (one with the original code, the other with the fault tolerance checks included) were set to unify two separate sets, each containing 1000 integers. The algorithm was run roughly 5000 times on both the original and the fault-tolerant code, injecting a fault into a different place in the code each time. The next two trials were to study the effects on a larger data size. These trials were set to unify two separate sets, each containing 100,000 integers, and ran the algorithm approximately 10,000 times.

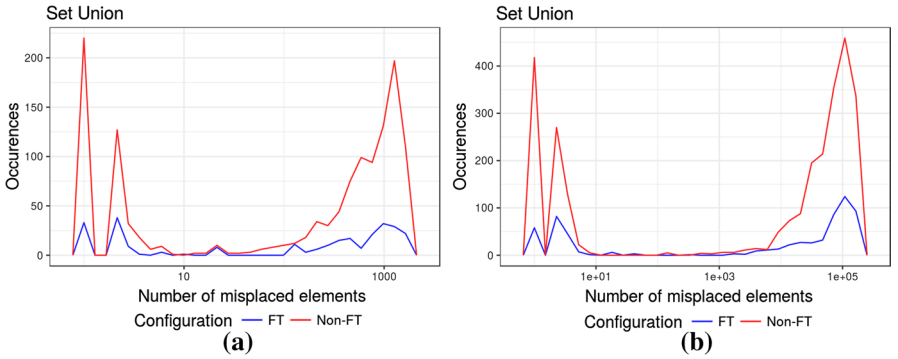

Fig. 17 Number of incorrect results by magnitude of error in incorrect outcomes in set union. **a** Small input. **b** Large input

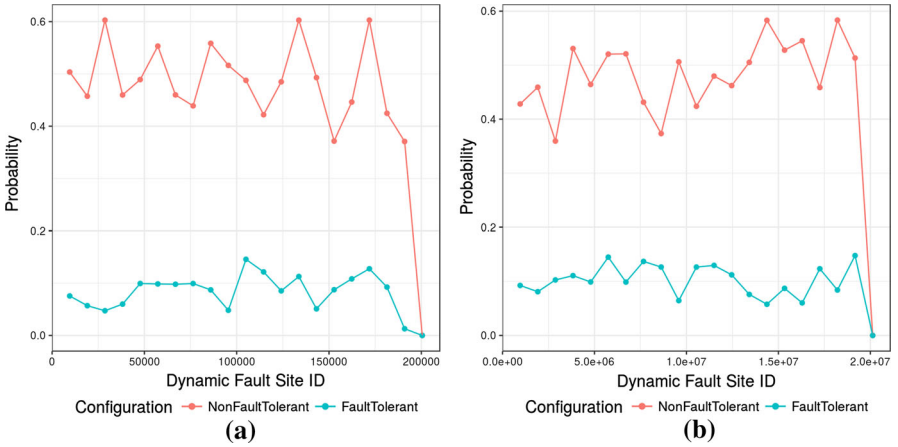

Fig. 18 Probability of error in non-fault-tolerant versus fault-tolerant runs of set union. **a** Small input. **b** Large input

4.6.4 Results

For the small data set, which was run roughly 5000 times, the number of errors in the non-fault-tolerant version versus the fault-tolerant version dropped from 1302 to 266. The large data set returned similar results but out of around 10,000 runs, showing a drop in faulty outcomes from 2672 to 651.

Error occurrences The two images in Fig. 17 are histograms of the number of faulty runs from least faulty to most faulty. This was determined by how many elements of the outputted union array were in the correct spot in the array. As shown in the graphs, the majority of the runs had either only a small amount of numbers wrong or almost all of the numbers wrong, which resulted in the big dip in the middle of the graphs.

The two graphs in Fig. 18 were generated to show the probability of error in the original code versus the fault-tolerant code. The *Y*-axis is the number of incorrect outputs divided by the total number of runs in the trial. The *X*-axis shows where in the

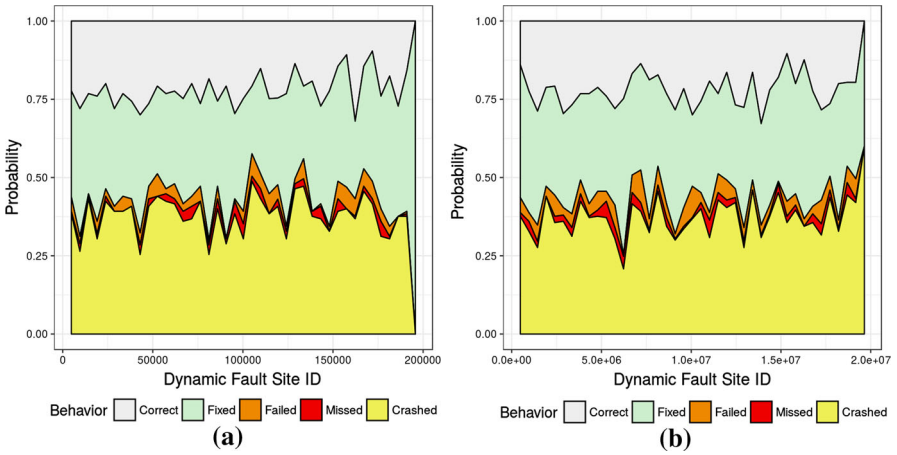

Fig. 19 Probability of different outcomes of the fault checker in set union. **a** Small input. **b** Large input

program the fault was injected (the left end being the beginning of the program, the right being the end). Both graphs for the large and small input do not seem to show much difference. The error probability of the non-fault-tolerant runs seems to fluctuate around 50%, while the fault-tolerant runs drop to around only 10% probability of error.

Behavior distribution Both of the bar graphs in Fig. 19, which represent the small and large input trials, seem to be fairly similar and do not exhibit any major differences. In both graphs, roughly 35% of the trials crashed due to some sort of error caused by fault injection. Another 35% of the runs resulted as "fixed" which shows that 35% of the runs were likely to return an incorrect answer but were corrected by the fault tolerance checks. The graph also shows that around 25% of the runs resulted in correct outputs (i.e., these runs were not affected by fault injection). The remaining 5% consists mostly of runs that resulted with incorrect outputs that were detected by the fault checks, but the fault checks failed to fix them. An extremely small number of runs ($\sim$ 1%) resulted as "missed," indicating that no error was detected but the output was incorrect, meaning a fault that altered the output data somehow slipped through the cracks of the fault checks.

4.7 Sort (Quicksort)

4.7.1 Introduction

Quicksort is known for being among the fastest sorting algorithms around. With an average run time of $O(n \cdot log(n))$, assuming n values, quicksort is often among the fastest sorting algorithms. Quick sort uses a recursive divide and conquer method, starting with a "pivot" number in an array and splitting it into two separate arrays, one containing all numbers larger than the pivot and the other containing all numbers smaller than the pivot. This is done recursively, selecting pivots in both of the smaller arrays and breaking them down further until the entire array is in order.

```
temp = data[left];
if (temp != data[left]) {
    cout << "Fault detectedn";
    temp = data[left];
```

(a)

```
label:
    while (left < right) {
        ...
    }
    if (left < right) {
        cout << "Fault detectedn";
        goto label;
```

(b)

```
label2:
int tmp = right - 1;
right --;
if (tmp != right) right = tmp;
```

(c)

```
label2:
if (left < right) {
    ...
} else {
    if (left < right) {
        ...
```

(d)

Fig. 20 Code examples in quicksort. **a** Assignment, **b** while loops, **c** increment, **d** if statements

4.7.2 Fault tolerance analysis

A large portion of the function is mainly conditional statements contained within `while()` and `if()` statements, so the most effective way to make this sorting algorithm fault-tolerant is to double check that these conditional statements were handled correctly/without error. To reduce these errors, four types of error checkers were used.

Assignments After each assignment statement in the code, an if statement, shown in Fig. 20a, was placed to assure that the data had been properly copied from source to destination. If not, a second attempt to assign the data is made. This type of checker is used four times after the following lines of code:

- `temp = data[left];`
- `data[left] = data[right];`
- `data[right] = data[left];`
- `data[left] = temp;`

While loops After each while statement in the code, an if statement, shown in Fig. 20b, was used to determine whether or not the while loop had been accurately terminated. If the condition in the if statement passes as true, then the program is set back to the beginning of the while loop so the loop can continue as it was supposed to. This checker was implemented into the program after three while loops:

- `while(left<right);`
- `while(left<right && data[right]>=temp);`
- `while(left<right && data[left]<=temp);`

Increment/decrement operators For each increment/decrement operator in the program, a temporary variable, shown in Fig. 20c, was created to assure the proper value

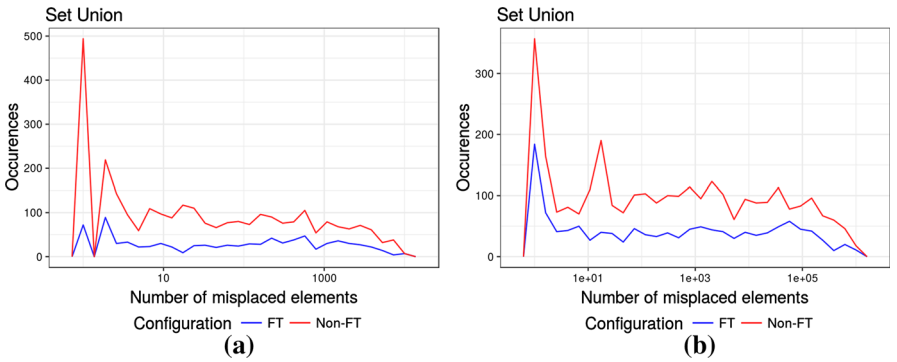

Fig. 21 Number of incorrect results by magnitude of error in incorrect outcomes in quicksort. **a** Small input. **b** Large input

was assigned to the variable. This checker was used twice in the program for the right– and left++ lines.

If statements Finally, an `else` statement, shown in Fig. 20d, was placed after each `if` statement in the program and within the else statement is a copy of the `if` statement as well as the lines of code that are within the brackets of the `if` statement. This assures that if the conditional statement was faulty in the first call of the `if` statement, the program will still execute the proper lines after it double checks the conditional statement in the else brackets. This checker is placed in the program three times after all three if statements that are `if(left<right)`.

4.7.3 Testing procedure

In order to show the effectiveness of the fault-tolerant version of the program versus the original code, we ran four separate trials. The first two trials were run on an input size of 10,000 values (one trial without fault-tolerant changes and the other with these fault-tolerant changes), while the other two trials were run on a size of 1,000,000 values to give an idea of how the size of the input may affect the fault tolerance of the program. Faults were injected into roughly 20,000 fault sites throughout each program giving us 20,000 different outputs for each trial.

4.7.4 Results

The small data set showed a drop in incorrect outcomes from 2822 to 854, while the large data set dropped from 3019 to 1289. This may indicate that as input data size increases, the program is less tolerant of faults and more prone to incorrect outcomes.

Error occurrences The graphs pictured in Fig. 21 display the frequency of varying error magnitudes in the incorrect runs. In all trials, the majority of the faulty runs seem to have a small faultiness value which results in the spike at the beginning of each graph. After the initial spike, the number of runs based on faultiness seems to be fairly consistent across the rest of the graph with a final dip at the end. In both graphs, the line generated by the fault-tolerant version clearly follows the non-fault-tolerant line

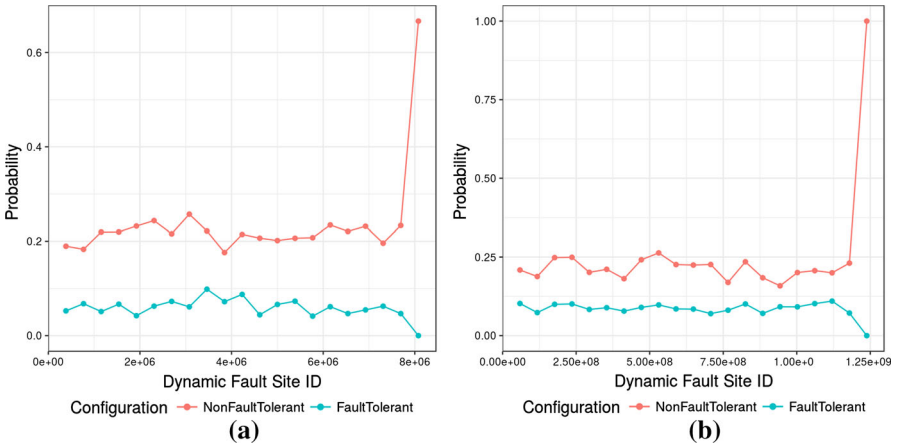

Fig. 22 Number of incorrect results by magnitude of error in incorrect outcomes in quicksort. **a** Small input. **b** Large input

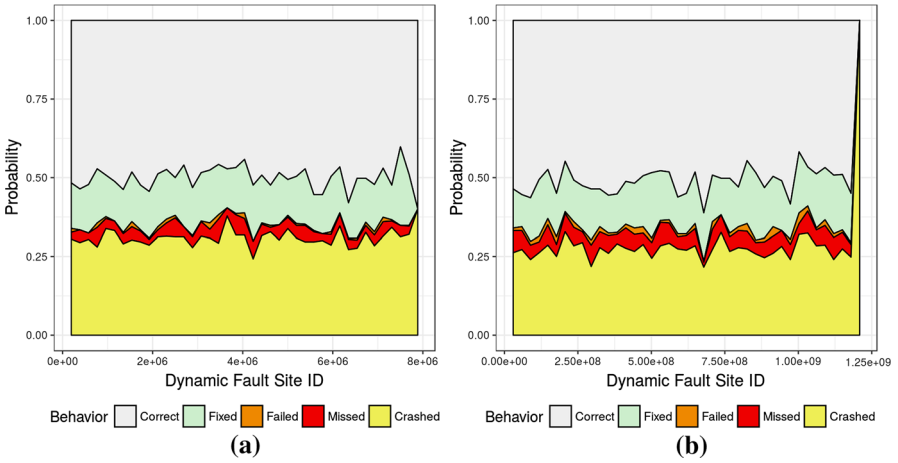

Fig. 23 Probability of the different outcomes of the fault checker in quicksort. **a** Small input. **b** Large input

but at almost half the size, showing that the fault tolerance checks were successful a significant portion of the time. The fault tolerance checks also seemed to remove a significant amount of the small errors in the small input, but only eliminated around half of these small errors in the large input indicating that the fault checks may be slightly less effective as input size grows.

Error probability The graphs in Fig. 22 do not take into account the amount of segmentation faults in the trial results. These graphs only consider runs of the program that were executed to completion, and display the ratio of incorrect results to overall number of outcomes in each given trial. In both cases, the percentage of incorrect outcomes seems to drop by roughly 15–20% due to the fault tolerance checks.

Behavior distribution The graphs in Fig. 23 do not seem to be different from each other by a noteworthy degree. Both graphs show that roughly 25% of the trials resulted in

segmentation faults, approximately 15–20% were "fixed" which shows that the fault tolerance correct a significant portion of runs that would have otherwise returned an incorrect result, and just over 50% of the runs turned up as correct which shows that over half of the faults injected had no effect on the outcome of the program. The majority of the remaining 5% ended up as "missed" with a small amount of "failed" runs as well.

4.8 Statistics operations (GREP)

4.8.1 Introduction

The acronym "grep" stands for "globally search for a regular expression and print." As the name suggests, the generic grep algorithm searches a given file or directory for a given regular expression and prints all found results. This specific grep implementation, MPI_Grep, prints the total number of found occurrences of the regular expression. It is actually a relatively simple algorithm, relying heavily on the C string library functions for searching.

4.8.2 Invariants

In order to identify and count all of the occurrences of a regular expression, grep divides up the given file into sections of a certain size, searches each section for the number of occurrences, and then aggregates the count from each section into a total count. As such, the invariants deal with both the counter variables and the pointers specifying locations within the file. These invariants were discovered by examining the algorithm and through testing. The specific invariants used are listed below:

In the aggregation function, let C_i be the count at iteration i. Then, $C_i + 1 \geq Ci$ and $C_i \geq 0$ for all iterations i. This can be observed from the code, as there is only an addition operation of two positive numbers per iteration, and it would not make sense to have a negative count of occurrences.

In the matching function, let C_i be the count at iteration i. Then, $C_i + 1 \geq C_i + 1 \geq C_i$ and $C_i \geq 0$ for all iterations i. Similar to above, the code only contains an addition of positive numbers. The added constraint is specified because the code only allows for one increment ("++") operation per iteration. If the counter were to increase by more than one per iteration, it would signal an error.

Let P_i be the location in the text at iteration i. Then, $P_i + 1 \geq P_i$ for all iterations i. This was discovered by looking at the string library "find" function, which searches the string for a given pattern in a forward direction. As such, it would not make sense for the location in the file to be a lower value after an iteration than it was previously.

4.8.3 Testing procedure

To test the grep algorithm's behavior under fault injection, we ran the grep algorithm on both a 52.4 and 522.2 MB file of random text. Both the fault-tolerant and non-fault-tolerant algorithm were run for a total of 10,000 outcomes on the file,

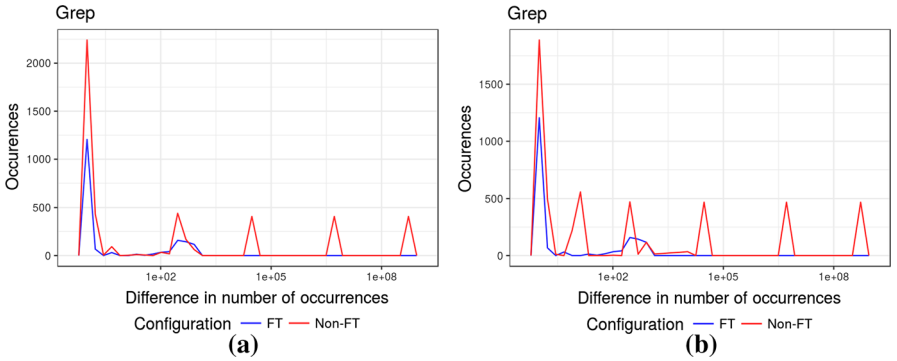

Fig. 24 Number of incorrect results by magnitude of error in incorrect outcomes in Grep. **a** Small input. **b** Large input

searching for the same pattern. Faults were injected into both the fault-tolerant and non-fault-tolerant code, ranging over the full length of possible fault sites. These faults could be injected into the bit positions 1, 8, 15, 22, or 29. Error in a given outcome was measured by the magnitude of the difference between found occurrences of the given outcome and the correct number of occurrences, as determined by a non-fault-tolerant, non-fault-injected MPI_Grep algorithm run on the same file with the same pattern.

4.8.4 Results

Error occurrences Out of the 10,000 outcomes for each version on the small input file (52.4 MB), the non-fault-tolerant version has a total number 4730 wrong counts and 232 segmentation faults, while the fault-tolerant version has only 1829 wrong counts and 139 segmentation faults. This shows an overall decrease in wrong counts by approximately 61.3% and a decrease in segmentation faults by 40.1%. The fault-tolerant version also boasts a much lower average difference from the correct count, with an average of 23 compared to the non-fault-tolerant versions average of 22,545,776.

Out of the 10,000 outcomes for each version on the large input file (522.2 MB), the non-fault-tolerant version has a total number 5298 wrong counts and 209 segmentation faults, while the fault-tolerant version has only 1804 wrong counts and 116 segmentation faults. This shows a similar improvement, with an overall decrease in wrong counts by approximately 65.9% and a decrease in segmentation faults by 44.5%. The fault-tolerant version again has a much lower average difference from the correct count, with an average of 347 compared to the non-fault-tolerant versions average of 25,864,037.

The first thing to note is that both the small and large input files exhibit a large number of errors having a count difference of only one shown in Fig. 24, though the non-fault-tolerant version reduces this number by approximately half. Next is that the non-fault-tolerant version seems to follow a certain pattern of error not present in the fault-tolerant version. In fact, the error values coincide directly with the bits

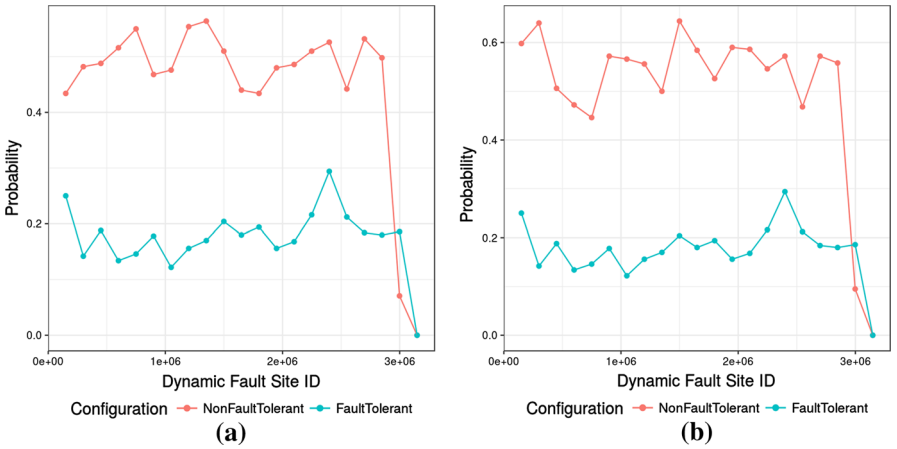

Fig. 25 Probability of error in non-fault-tolerant versus fault-tolerant runs of Grep. **a** Small input. **b** Large input

into which we are injecting errors. Since we are injecting into bits 1, 8, 15, 22, and 29, the error values are interestingly enough 2E1, 2E8, and so forth. This shows that the output of the program is very sensitive to single bit-flip errors that may occur. This also explains why the non-fault-tolerant version has such a large mean error. However, it would seem that the fault-tolerant version introduces error patterns not already present in the non-fault-tolerant version or exacerbates errors already present. This can be due to errors being injected into the fault-tolerant code itself, resulting in unexpected behavior.

Error probability From these graphs in Fig. 25, it is clear that the fault-tolerant program, at all stages of execution (ignoring the drop at the end of execution this is caused by the difference in dynamic fault site count), largely reduces the probability of error by over half, with a greater improvement being seen in large input files. This is in agreement with the previous histograms, which show a large reduction in the count of errors (and thus the probability of error).

Program behavior The first thing to notice in Fig. 26 is a very large percentage of fixed outcomes, with both large and small input files. This shows that many of the detected errors are correctly fixed to result in proper program behavior. There is also a substantial number of correct outputs, meaning that the faults injected were not seen by the fault-tolerant code to affect program output. With both input files, there is a moderate number of missed errors—this means that some faults are injected that do end up affecting the output and have not been accounted for. There is a small percentage of failed outcomes as well, more so with large input files. Thus, larger input files have more injected faults that it can detect but cannot correct properly. Lastly, there are a small percentage of outcomes that crash and thus fail to produce outcome at all. This seems to be handled better with large input files.

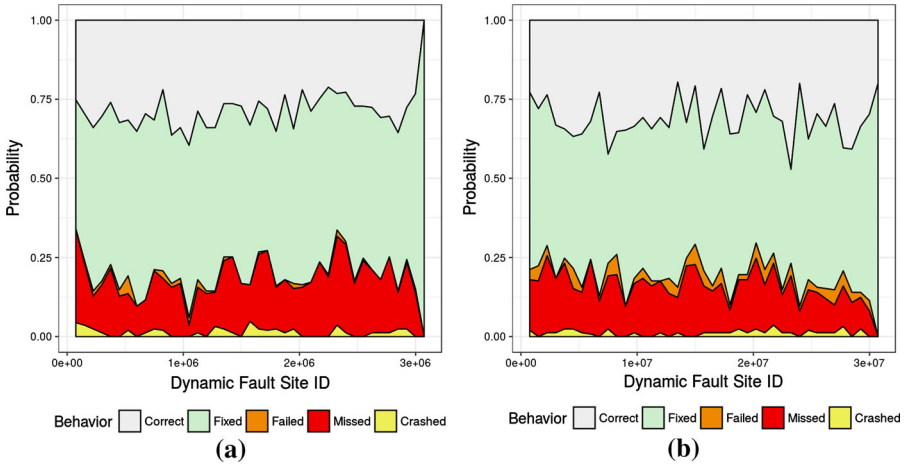

Fig. 26 Probability of different outcomes of the fault checker in Grep. **a** Small input. **b** Large input

5 Determining the number of experiments

The sample space (total number of dynamic fault sites) can be prohibitively large. To be practical, we sample a small fraction of the space, ensuring that the drawn sample is statistically sound and is representative of the fault injection space. For a fault-injected run, its outcome is a direct result of the chosen fault injection parameters, namely the dynamic/static fault site IDs and the bit IDs. Therefore, a sample that is representative of the fault site should retain the relation between dynamic/static fault sites to the outcomes.

To quantify the quality of the sample, we train a boosted regression tree model as a proxy, as this model can discover the relation between fault site/bit IDs and the outcome. A sample that is good should enable the model to report its results with a high confidence level, which translates to a low variance in its prediction accuracy. We measure the accuracy of the model as well as the variance in its accuracy by cross-validation on the collected sample.

Figure 27 shows the trend of the standard deviation (square root of variance) and means of the accuracy of the regression model with regard to the increasing sample size for each of the kernels. It can be observed from the figure that:

- The expected accuracy of the model shows an increasing trend as the number of experiments increases and stabilizes after a certain point. In comparison, randomly guessing one of three possible outcomes will result in an accuracy of 33%.
- The standard deviation of the model accuracy decreases as the number of experiments increases, suggesting the model's increasing confidence in its prediction. The standard deviation is a result of two sources: 1) the instability of the model and 2) the finiteness of the sample. As the sample size increases, variance resulting from sampling will decrease and is reflected in the decrease in the overall standard deviation of the model accuracy.

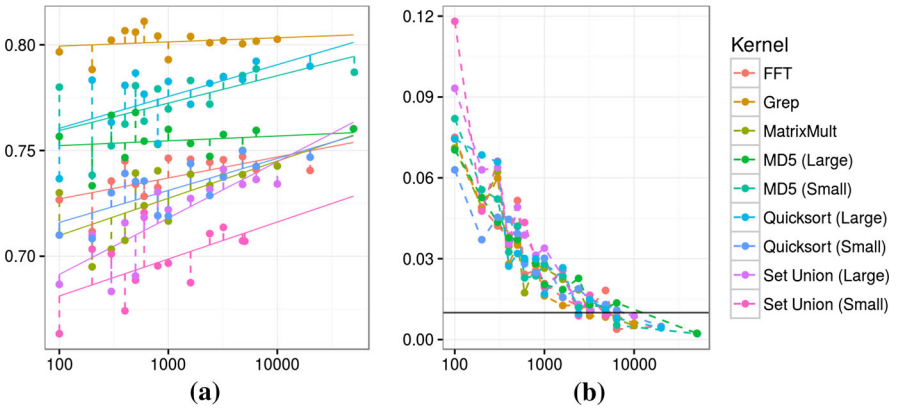

Fig. 27 **a** Expected accuracy. **b** Standard deviation. Trend of the expectation (*left*) and standard deviation (*right*) of the accuracy of the boosted regression tree classifier trained on samples of varying sizes (*X* axis). The threshold on standard deviation used to determine experiment size (0.01) is marked by the *black line* in (**b**)

Based on these observations, we choose the number of experiments such that the standard deviation in the accuracy of the model built on the sample decreases below a threshold (in this experiment, 0.01), to make sure that the sample is large enough for analytic models to produce results with high confidence and narrow error intervals.

6 Comparison of dwarves

Each of the dwarves included in the previous section displays a reduction in wrong answers and some additionally lowers the execution failure counts. However, some display much greater improvement than others. In this section, we aim to summarize the results of all eight dwarves and provide explanation on their relative behaviors.

Figure 28 includes a summary of all eight dwarves. The columns on the left detail the non-fault-tolerant performance results of algorithms, while the columns on the right display the performance outcomes of their respective fault-tolerant versions. This allows for an investigation of the improvement of each algorithm while making comparisons between algorithms simple.

To begin, some algorithms provided higher-level invariants that allow for simpler check systems, such as matrix multiplication and FFT (Type **1**). These algorithms display the greatest vulnerability to faults, which we suspect is caused by their reliance on complex data structures and the propagation of error through the program. However, the check systems based on high-level invariants manage to identify and fix the majority of wrong answers and can additionally avoid abnormal termination. We believe this is due to the reliance on the high-level invariants for error identification and correction that detect errors before they propagate into program termination. Therefore, this first collection of dwarves with high-level invariants displays the least error resilience natively but the greatest improvement once fault-tolerant additions are included.

Other algorithms such as grep and set union lacked higher-level invariants, relying on lower-level checking systems (see Fig. 16 for examples). These algorithms display

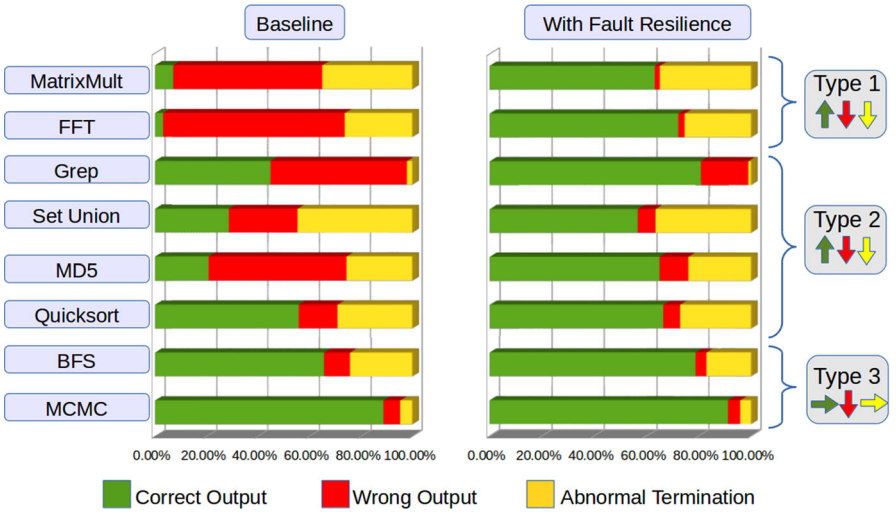

Fig. 28 Aggregated data of all eight dwarves. Non-fault-tolerant behavior shown on *left*, fault-tolerant behavior on *right*

moderate innate fault tolerance and respond reasonably well with the added fault-tolerant mechanisms. The improvement is less than that of the first group of dwarves, but still yields a reduction in both wrong answers and program failure occurrences.

Lastly, BFS and MCMC (Type **3**) both do not show much improvement in their performance. They display minor reductions in wrong answers and program failure outcomes, but at a much smaller rate than the previous groups. Additionally, they both display relatively large correct answer percentages without the addition of fault-tolerant mechanisms. We attribute this to the design of the algorithms themselves. For instance, MCMC relies on sampling a random distribution to construct the original distribution. If an error is injected into the generation of a single random value, that is not easily detectable nor does it greatly influence the behavior of the algorithm as there are many other samples taken that are not faulty.

To combine with these aggregated results, we measured the execution time of the non-fault-tolerant and fault-tolerant versions of each algorithm to compute execution overhead percentages (Fig. 29). This was measured by execution both versions of each algorithm fifty times without injecting faults and averaging the collected execution times of each algorithm. As would be expected, the algorithms with higher-level invariants (type 1 algorithms in Fig. 28) have low overhead percentages, while those with lower-level checkers (type 2 and type 3 algorithms in Fig. 28) display greater overhead percentages. Most interesting is the large overhead of MD5, which suggests that the fault-tolerant additions add more overhead than its improvement warrants.

To this end, we calculated expected running times of each algorithm and normalized the results by the measured running times of the non-fault-tolerant versions. This calculation is essentially a weighted average for each algorithm, weighing the fault-tolerant and non-fault-tolerant execution times by their error probabilities. This allows one to compare the algorithms by their expected execution times to view which algorithms display the best improvement rates for their respective overhead values, as shown in

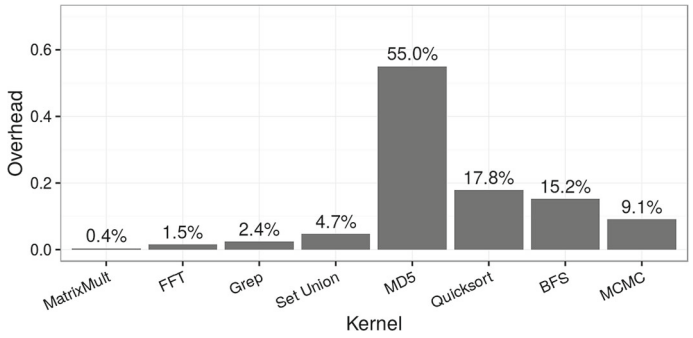

Fig. 29 Measured execution time overhead percentages for each algorithm

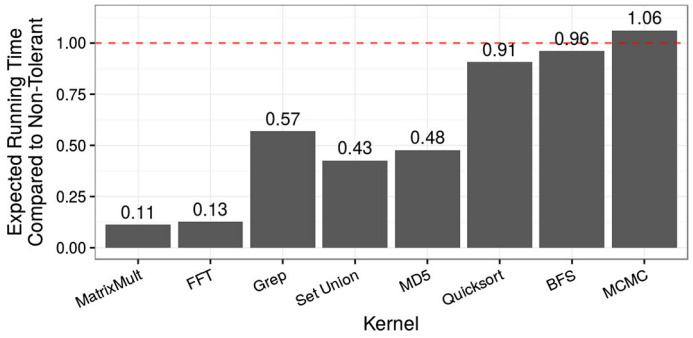

Fig. 30 Normalized expected running times for each algorithm

Fig. 30. We have drawn a line to mark the baseline non-fault-tolerant performance of each algorithm.

Viewing these expected execution times, the algorithms group themselves similarly to Fig. 28. We see that the Matrix Multiplication and FFT algorithms display the lowest running times and thus the greatest improvement for the introduced overhead. What is surprising is the relative improvement of the type 2 algorithms, namely MD5. Despite the relatively large overheads, the large reductions in error probability still result in an improved expected running time. It also seems that, taking overhead into account, quicksort aligns more closely with the type 3 algorithms than the type 2 algorithms. These type 3 algorithms all display relatively minor expected running time improvements, even exceeding the baseline performance in the case of MCMC. This means that for the type 3 algorithms, the fault-tolerant additions are not very beneficial for the overhead they introduce, while type 1 algorithms show large improvement and type 2 algorithms show moderate improvement.

7 Conclusion

We have shown the effects of fault-tolerant code added into Big Data algorithms by experimenting with eight Big Data Dwarves as defined by the Big Data Benchmark

suite. For each of the eight dwarves, we have implemented algorithm-specific invariants where applicable to identify and correct errors in program execution. We have discussed the effects of the fault-tolerant additions to each algorithm individually by evaluating error magnitudes, error probabilities, and program output behavior. Additionally, we have compared the eight algorithms to each other, resulting in three classes of Big Data dwarves: those with high-level invariants that are extremely vulnerable to faults, but show the most improvement with fault-tolerant code additions; those without high-level invariants but with moderate natural resilience, which show lower improvement rates than the first group; and those that are naturally extremely resilient, which show minor execution improvement with the fault-tolerant code additions. We have also analyzed the overhead introduced by the fault-tolerant mechanisms and quantified the expected running times for the algorithms to evaluate the benefit of the mechanisms in light of their overhead introduced, which supports the fault-tolerant performance of algorithms being grouped by level of invariant.

Together, these analyses create a portfolio displaying the resilience that fault-tolerant additions can lend to common Big Data algorithms, reducing the chances of program failure and wrong output. This additionally reduces the time and energy wasted to rerun faulty algorithms and helps avoid the danger of not detecting a wrong output which, if undetected, could cause unprecedented damage.

8 Future works

We aim to expand our research to include similar tests on parallel applications. We believe this will more accurately represent realistic usage of Big Data applications and error resilience as most of these programs would rely on parallelism. Additionally, we hope to expand the functionality of KULFI to include instruction encoding errors, again to more realistically simulate natural soft faults.

Acknowledgements We are grateful to Prof. Nian-Feng Tzeng at the Center for Advanced Computer Studies, University of Louisiana at Lafayette, for providing invaluable feedbacks to our research. We are grateful to Vishal Sharma and Arvind Haran, the authors of the original KULFI and for granting us permission to modify it for our experiment purposes. We are also appreciative of the opportunity to be involved in and contribute to KULFI. Support of this research was provided by National Science Foundation under Award Numbers: 1527318, 1422408 (Directorate for Computer and Information Science and Engineering), and 1017961 (Division of Computing and Communication Foundations).

References

1. Christoforides A (2011) Metropolis–Hastings implementation. https://github.com/alexischr/mh
2. IBM's Big Data Platform and Decision Management (2012) What is big data? http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html
3. Gordon R (2013) http://math.stackexchange.com/questions/346894/prove-of-the-parsevals-theorem-for-discrete-fourier-transform-dft
4. Sharma V, Haran A, Chen S (2013) Kulfi fault injector. http://github.com/quadpixels/kulfi
5. AMPLab at University of California, Berkeley (2014) AMPLab big data benchmark. https://amplab.cs.berkeley.edu/benchmark/
6. Harris M, NVidia (2015) https://devblogs.nvidia.com/parallelforall/new-features-cuda-7-5/

7. Armstrong TG, Ponnekanti V, Borthakur D, Callaghan M (2013) Linkbench: a database benchmark based on the Facebook social graph. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD '13, pp 1185–1196. doi:10.1145/2463676.2465296
8. Austin T (1999) Diva: a reliable substrate for deep submicron microarchitecture design. In: Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO 1999)
9. Bender C, Sanda PN, Kudva P, Mata R, Pokala V, Haraden R, Schallhorn M (2008) Soft-error resilience of the ibm power6 processor input/output subsystem. IBM J Res Dev 52(3):285–292. doi:10.1147/rd.523.0285
10. Cappello F, Geist A, Gropp W, Kale S, Kramer B, Snir M (2014) Toward exascale resilience: 2014 update. J Supercomput Front Innov 1(1). doi:10.14529/jsfi140101
11. Chen S, Bronevetsky G, Li B, Guix MC, Peng L (2015) A framework for evaluating comprehensive fault resilience mechanisms in numerical programs. J Supercomput 71(8):2963–2984. doi:10.1007/s11227-015-1422-z
12. Chen S, Bronevetsky G, Peng L, Li B, Fu X (2016) Soft error resilience in big data kernels through modular analysis. J Supercomput 72(4):1570–1596. doi:10.1007/s11227-016-1682-2
13. Chung J, Lee I, Sullivan M, Ryoo JH, Kim DW, Yoon DH, Kaplan L, Erez M (2012) Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC12)
14. Collange S, Defour D, Graillat S, Iakymchuk R (2015) Numerical reproducibility for the parallel reduction on multi- and many-core architectures. Parallel Comput 49:83–97. doi:10.1016/j.parco.2015.09.001, http://www.sciencedirect.com/science/article/pii/S0167819115001155
15. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, New York, NY, USA, SoCC '10, pp 143–154, doi:10.1145/1807128.1807152
16. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge, MA
17. Ferdman M, Adileh A, Koçberber YO, Volos S, Alisafaee M, Jevdjic D, Kaynak C, Popescu AD, Ailamaki A, Falsafi B (2012) Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3–7, 2012, pp 37–48. doi:10.1145/2150976.2150982
18. Free Software Foundation (2016) GSL—GNU scientific library. https://www.gnu.org/software/gsl/
19. Gao W, Luo C, Zhan J, Ye H, He X, Wang L, Zhu Y, Tian X (2015) Identifying dwarfs workloads in big data analytics. http://arxiv.org/abs/1505.06872
20. Ghazal A, Rabl T, Hu M, Raab F, Poess M, Crolotte A, Jacobsen HA (2013) Bigbench: towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, SIGMOD '13, pp 1197–1208. doi:10.1145/2463676.2463712
21. Guan Q, Debardeleben N, Blanchard S, Wu P, Monrow L, Chen Z (2016) P-FSEFI: a parallel soft error fault injection framework for parallel applications. In: Proceedings of the 12th Workshop on Silicon Error in Logic-System Effect (SELSE)
22. Huang S, Huang J, Dai J, Xie T, Huang B (2010) The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW), pp 41–51. doi:10.1109/ICDEW.2010.5452747
23. Iakymchuk R, Collagne S, Defour D, Graillat S (2015) Exblas: reproducible and accurate BLAS library. In the Proceedings of the Numerical Reproducibility at Exascale (NRE2015) workshop held as part of the Supercomputing Conference (SC15). Austin, TX, USA, November 15-20, 2015. HAL ID: hal-01202396
24. ITRS (2013) International technology roadmap for semiconductors. Technical report
25. Kumar S, Hari S, Adve SV, Naeimi H, Ramachandran P (2012) Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In: Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)

26. Lattner C, Adve V (2004) LLVM: A compilation framework for lifelong program analysis and transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004), San Jose, CA, USA
27. Liu W, Zhang W, Wang X, Xu J (2016) Distributed sensor network-on-chip for performance optimization of soft-error-tolerant multiprocessor system-on-chip. IEEE Trans Very Large Scale Integr (VLSI) Syst 24(4):1546–1559. doi:10.1109/TVLSI.2015.2452910
28. NVIDIA (2013) Tesla k20 gpu accelerator. http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v07.pdf
29. Serrano F, Clemente JA, Mecha H (2015) A methodology to emulate single event upsets in flip-flops using FPGAs through partial reconfiguration and instrumentation. IEEE Trans Nucl Sci 62(4):1617–1624. doi:10.1109/TNS.2015.2447391
30. Tiwari D, Gupta S, Gallarno G, Rogers J, Maxwell D (2015) Reliability lessons learned from GPU experience with the titan supercomputer at oak ridge leadership computing facility. In: SC15: International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–12. doi:10.1145/2807591.2807666
31. Wang L, Bertran R, Buyuktosunoglu A, Bose P, Skadron K (2014) Characterization of transient error tolerance for a class of mobile embedded applications. In: 2014 IEEE International Symposium on Workload Characterization (IISWC), pp 74–75. doi:10.1109/IISWC.2014.6983042
32. Yeh TY, Reinman G, Patel SJ, Faloutsos P (2009) Fool me twice: exploring and exploiting error tolerance in physics-based animation. ACM Trans Graph 29(1):5:1–5:11. doi:10.1145/1640443.1640448