Correcting Soft Errors Online in Fast Fourier Transform

Xin Liang
University of California, Riverside
Riverside, CA 92521
xlian007@ucr.edu

Jieyang Chen
University of California, Riverside
Riverside, CA 92521
jchen098@ucr.edu

Dingwen Tao
University of California, Riverside
Riverside, CA 92521
dtao001@ucr.edu

Sihuan Li
University of California, Riverside
Riverside, CA 92521
sli049@ucr.edu

Panruo Wu
University of California, Riverside
Riverside, CA 92521
pwu011@ucr.edu

Hongbo Li
University of California, Riverside
Riverside, CA 92521
hli035@ucr.edu

Kaiming Ouyang
University of California, Riverside
Riverside, CA 92521
kouya001@ucr.edu

Yuanlai Liu
University of California, Riverside
Riverside, CA 92521
yliu158@ucr.edu

Fengguang Song
Indiana University-Purdue University Indianapolis
Indianapolis, IN 46202
fgsong@cs.iupui.edu

Zizhong Chen
University of California, Riverside
Riverside, CA 92521
chen@cs.ucr.edu

ABSTRACT

While many algorithm-based fault tolerance (ABFT) schemes have been proposed to detect soft errors offline in the fast Fourier transform (FFT) after computation finishes, none of the existing ABFT schemes detect soft errors online before the computation finishes. This paper presents an online ABFT scheme for FFT so that soft errors can be detected online and the corrupted computation can be terminated in a much more timely manner. We also extend our scheme to tolerate both arithmetic errors and memory errors, develop strategies to reduce its fault tolerance overhead and improve its numerical stability and fault coverage, and finally incorporate it into the widely used FFTW library - one of the today's fastest FFT software implementations. Experimental results demonstrate that: (1) the proposed online ABFT scheme introduces much lower overhead than the existing offline ABFT schemes; (2) it detects errors in a much more timely manner; and (3) it also has higher numerical stability and better fault coverage.

KEYWORDS

Algorithm-Based Fault Tolerance, Soft Errors, DFT, FFT, FFTW

ACM Reference format:
Xin Liang, Jieyang Chen, Dingwen Tao, Sihuan Li, Panruo Wu, Hongbo Li, Kaiming Ouyang, Yuanlai Liu, Fengguang Song, and Zizhong Chen. 2017. Correcting Soft Errors Online in Fast Fourier Transform. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 12 pages.
DOI: 10.1145/3126908.3126915

SC17, Denver, CO, USA

DOI: 10.1145/3126908.3126915

1 INTRODUCTION

As the size of transistors continues to reduce and the number of components continues to increase, soft errors in supercomputers become more and more common [18]. In fault tolerance literature, many techniques have been proposed to detect and/or correct soft errors. The best-known general technique to detect soft errors is the double modular redundancy (DMR) approach. This approach either uses two different hardware units to perform the same computation at the same time or performs the same computation on the same hardware twice, then compares the two results to detect whether errors occur or not. The most well-known general technique to correct single soft errors is the triple modular redundancy (TMR) approach. TMR either performs the same computation on three different hardware units or uses the same hardware to perform the same computation for three times, then compares and votes the majority results as the correct result. While DMR and TMR are very general, their overhead is high - at least 100% overhead to detect errors and 200% overhead to correct errors.

To protect memory corruption, ECC (Error Correcting Codes) memory has been widely used by many computer vendors. Although today's ECC memory can detect and correct bit flips in memory, it brings significant overhead in space, time, and energy. Furthermore, ECC memory is not able to handle computational (i.e. arithmetic) errors that are caused by faults in logic units.

In order to significantly reduce fault tolerance overhead, algorithmic characteristics have been leveraged to design highly efficient fault tolerance schemes since 1984 [20]. Over the past thirty years, many algorithm-based fault tolerance (ABFT) schemes have been

proposed in literature. In [20], Huang and Abraham proposed the first ABFT scheme to detect miscalculations in matrix operations on systolic arrays. In [3], Banerjee et al. proposed an ABFT scheme that works for hypercube multiprocessor. In [29, 30], Shantharam et al. analyzed the impact of soft errors on iterative linear algebra methods and proposed a fault tolerant scheme for preconditioned Conjugate Gradient methods for sparse linear systems. In [31], Sloan et al. propsoed an algorithmic approach to detect errors in sparse linear algebra. In [28], Sao and Vuduc explored a self-stabilizing fault tolerance approach for iterative methods. In [14], Elliott et al. analyzed the impact of soft errors on GMRES algorithm. In [24], Li et al. designed an ABFT scheme with a cooperative software-hardware approach. In [37, 39], Wu et al. proposed an ABFT scheme to correct errors in matrix operation online. In [12], Davies proposed an online scheme to correct soft errors in LU factorization. In [13], Di and Cappello carefully characterized 18 real-world HPC applications and proposed an adaptive impact-driven approach to detect errors in these applications. In [9], Chien et al. proposed a new programming approach GVR that allows applications to describe error detection (checking) and recovery routines and inject them into the GVR stack for efficient implementation. In [4], Bridges et al. proposed a fault-tolerant linear solvers via selective reliability. In [32], Stoyanov and Webster showed some numerical analysis of fixed point algorithms for silent hardware faults. Besides those, some more work is carried out on linear algebra methods [5, 7, 8, 38, 40] , iterative solvers [21, 34], and error propagations [2, 6].

For fast Fourier transform (FFT), Antola et al. proposed a time-redundant scheme in [1]. In [10], Choi and Malek introduced a fault tolerance scheme for FFT that is based on recomputing through an alternate path. In [22], Jou and Abraham proposed an ABFT scheme for the FFT networks that can achieve 100% fault coverage and throughput at a cost of $O(\frac{2}{\log_2 N})$ hardware overhead. Later, in [33], Tao and Hartmann came up with a novel encoding scheme for FFT networks which has higher fault coverage by adding 5% hardware. After that, in [35], Wang and Jha presented a new concurrent error detection (CED) scheme that achieves better result with less hardware redundancy. Then, in [26], Oh showed a similar CED scheme using a different checksum with increased fault coverage. Additionally, some progress has also been made on parallel system and GPUs. Banerjee [3] proposed a fault tolerant design on hypercube multiprocessors. Pilla [27] presented specific software-based hardening strategies to reduce the failure rate. Fu and Yang [17] also implemented a fault tolerant parallel FFT using MPI.

While many offline ABFT schemes have been proposed for FFT over the past thirty years, a careful review of the existing ABFT literature indicates that no previous ABFT schemes can detect and correct soft errors online before an FFT computation finishes. This paper proposes an online ABFT scheme for FFT so that errors in an FFT computation can be efficiently corrected in the middle of the computing in a timely manner before the computation finishes. Because the FFT of a large vector is often computed via computing the FFTs of many smaller sub-vectors, a natural idea to correct errors online is to use the existing offline ABFT approaches to each small FFT computations. However, in this paper, we find that simply applying offline ABFT to each decomposed small FFTs introduces too much overhead due to the following facts. Firstly, the input of the decomposed FFTs is non-contiguous. Multiple non-contiguous reads or writes cause much longer memory access time because of heavy cache misses. Secondly, separated function calls to the fault tolerant version small FFTs would not reuse the computed input checksum vector, making the online version at least twice slower than the offline version. Thirdly, there will be at least three memory checksum generations and verifications since each divided FFT needs to be protected and there is a rearrangement of data after the first part, leading to large overhead when memory errors are taken into consideration.

FFT is widely used to compute the discrete Fourier transform (DFT). DFT plays a very important role in engineering, science, and mathematics. Therefore, reliable and fast computing of DFT will benefit not only a large number of people but also a wide range of fields. The main contributions of this paper include:

- **The first online ABFT scheme for FFT:** Existing ABFT schemes for FFT [1, 10, 22, 25, 26, 33, 35] detect soft errors offline after the FFT computation finishes. Even if an error occurs at the beginning of the FFT, existing ABFT schemes can not detect it in a timely manner, hence, have to allow the corrupted computation to continue until it finishes, then verify the correctness. After an error is detected, the whole FFT computation has to be restarted. This paper designs an online ABFT scheme that is able to detect errors online soon after the error occurs so that the corrupted computation can be terminated in a timely manner. After the corrupted computation is terminated, instead of repeating the whole computation from the beginning, the proposed online ABFT scheme only need to repeat a small fraction the computation. Therefore the computation efficiency will be greatly improved when errors occur.
- **The first soft-error-resilient FFT software implementation - FT-FFTW:** Existing FFT ABFT schemes are either designed for hard errors or designed under the context of hardware implementation. This paper develops soft-error-resilient FFT software for the first time. We develop FT-FFTW, incorporate both the existing offline ABFT and the newly proposed online ABFT into one of the today's fastest FFT software libraries - FFTW, and validate the implementations on TIANHE-2 supercomputer. Experimental results demonstrate that the proposed online ABFT is able to detect soft errors in a timely manner before the computation finishes and improve the computation efficiency by a factor of two when errors occur.
- **Innovative optimizations for online ABFT FFT:** It is very challenging to add fault tolerance capability to the highly optimized FFTW library without introducing significant performance penalty. Simply applying existing ABFT to each small FFTs within a large FFT introduces too much overhead. This paper develops several optimization strategies to reduce the overhead. The optimized online ABFT FFT introduces lower overhead than the existing offline scheme even if no error occurs.
- **The first online ABFT scheme for parallel in-place FFT:** Different from the out-of-place sequential FFT, the parallel FFT tends to use in-place FFT with no auxiliary space. We develop an online ABFT scheme for in-place FFT and extend our FFT ABFT scheme from sequential to parallel.

- **Parallel optimization strategy to minimize the overhead:** We develop a communication-computation overlap strategy to hide half of the fault tolerance cost for our parallel FT-FFTW. With the re-designed plan, the parallel FT-FFTW is able to achieve comparable performance to the original FFTW library.
- **Significant improvement in numerical stability and fault coverage:** Round-off errors for floating point calculations affect the numerical stability and fault coverage. This paper analyzes the impact of round-off errors for our online ABFT scheme in detail and shows that our online ABFT scheme has higher numerical stability and better fault coverage than the existing schemes.

When developing fault tolerance schemes, there is a trade-off between generality and efficiency. In order to leverage the algorithmic characteristics to optimize efficiency, this paper trades generality for better efficiency. While automating the proposed ABFT scheme to gain generality will loss the efficiency obtained, part of the idea in this paper can still be generalized to other divide-and-conquer applications if an offline fault tolerance scheme can be designed for each individual sub-problem.

2 BACKGROUND

2.1 DFT and FFT

The DFT for a complex sequence can be calculated as follows:

$$X_j = \sum_{n=0}^{N-1} x_n \omega_N^{jn}, j = 0, 1, \ldots, N-1 \quad (1)$$

where $\omega_N = \exp^{-i\frac{2\pi}{N}}$ and $i = \sqrt{-1}$ is the unit imaginary root. Correspondingly, the inverse discrete Fourier transform (IDFT) can be calculated as:

$$X_j = \frac{1}{N} \sum_{n=0}^{N-1} x_n \omega_N^{-jn}, j = 0, 1, \ldots, N-1$$

If DFT or IDFT is calculated directly, it is obvious that $O(N^2)$ operations are needed as each element costs $O(N)$ operations. To save more time, the fast Fourier transform (FFT) has been proposed to reduce the number of operations to $O(N \log N)$. The most popular Cooley-Tukey algorithm for FFT can be derived as follows. If the size N can be factorized into two smaller integers as $N = N_1 N_2$, (1) can be rewritten by letting $j = j_1 N_2 + j_2$ and $n = n_2 N_1 + n_1$:

$$X_{j_1 N_2 + j_2} = \sum_{n_1=0}^{N_1-1} \Big(\sum_{n_2=0}^{N_2-1} (x_{n_2 N_1 + n_1} \omega_{N_2}^{n_2 j_2}) \omega_N^{n_1 j_2} \Big) \omega_{N_1}^{n_1 j_1} \quad (2)$$

$\sum_{n_2=0}^{N_2-1} x_{n_2 N_1 + n_1} \omega_{N_2}^{n_2 j_2}$ is an N_2-point DFT and $\sum_{n_1=0}^{N_1-1}(\ldots)\omega_{N_1}^{n_1 j_1}$ is an N_1-point DFT. Thus the original N-point DFT is decomposed to N_1 inner DFTs of size N_2 and N_2 outer DFTs of size N_1. These N_1-point DFTs and N_2-point DFTs can also be decomposed into DFTs of smaller sizes recursively. By this means, the total operations of DFT is reduced to $O(N \log N)$.

2.2 Previous Fault Tolerant Work for FFT

Many ABFT schemes have been designed to detect and correct soft errors in FFT. These schemes typically use concurrent error detection scheme with encoding and decoding system. To illustrate how these ABFT schemes work, we take Wang's approach in [35] as an example. As a special case of matrix-vector multiplication, a DFT can be written into matrix form according to equation (1):

$$\begin{bmatrix} X(0) \\ X(1) \\ \vdots \\ X(N-1) \end{bmatrix} = \begin{bmatrix} \omega_N^0 & \omega_N^0 & \ldots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \ldots & \omega_N^{n-1} \\ \omega_N^0 & \omega_N^2 & \ldots & \omega_N^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{n-1} & \ldots & \omega_N^{(n-1)^2} \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ \vdots \\ x(N-1) \end{bmatrix}$$

Let A denote the coefficient matrix where $A_{ij} = \omega_N^{ij}$, X denotes the output vector, x denotes the input vector, the matrix form can be simply written as $X = Ax$. The equation maintains by multiplying X and Ax with a selected checksum vector r:

$$\begin{bmatrix} X \\ rX \end{bmatrix} = \begin{bmatrix} Ax \\ rAx \end{bmatrix}$$

r is called the weighted checksum for this matrix operation. The last row of the matrix can be expanded as:

$$\sum_{j=0}^{N-1} r_j X_j = \sum_{j=0}^{N-1} (rA)_j x_j$$

Then by comparing the results of the two checksums, any computational error can be detected.

However, not all checksum schemes are suitable for ABFT FFT. It has been proved in [35] that the following checksum scheme works well for ABFT FFT:

$$r = (\omega_3^0, \omega_3^1, \ldots, \omega_3^{N-1})$$

where $\omega_3 = -\frac{1}{2} + \frac{\sqrt{3}}{2} i$ is the first cube root of 1.

As for error correction, time redundancy methods are preferred in almost all the approaches. Re-calculation is necessary to produce the correct result.

All of these ABFT schemes mentioned above are proposed for hardware implementation. They assume that the size of input is fixed for a specific FFT implementation. In the hardware implementation, they detect errors by comparing the difference of rX and rAx, and they assume input checksum vector rA can be pre-calculated when output checksum vector r is given. However, software FFT implementations usually accept varying sizes of input, and thus extra overhead will be introduced to calculate rA. The software-level implementation of this approach is shown in Algorithm 1.

Algorithm 1 Offline ABFT FFT Algorithm

```
1: procedure OFFLINE-ABFT-FFT
2:     Set the calculation flag calcFlag = true
3:     Calculate input checksum vector c = rA
4:     while calcFlag do
5:         Calculate the FFT: X = Ax
6:         calcFlag = (|rX − cx| > η)
7:     end while
8: end procedure
```

3 ONLINE ABFT FFT SCHEMES

To correct errors in a more timely manner, two online schemes are proposed in this section. As faults are categorized into two

Algorithm 2 Online ABFT FFT Algorithm

procedure ONLINE-ABFT-FFT
  Get initial radix k and corresponding $m = \frac{N}{k}$
  Calculate input checksum vector $c_m = r_m A_m$ with DMR
  for i from 0 to $k-1$ **do**
    Set the calculation flag $calcFlag = true$
    while $calcFlag$ **do**
      Calculate the i-th FFT: $X'_i = A_m x_i$
      $calcFlag = (|r_m X'_i - c_m x_i| > \eta_1)$
    end while
  end for
  Calculate input checksum vector $c_k = r_k A_k$ with DMR
  for i from 0 to $m-1$ **do**
    Multiply twiddle factor: $X''_i = twiddle_i . * X'_i$ with DMR
    Set the calculation flag $calcFlag = true$
    while $calcFlag$ **do**
      Calculate the i-th FFT: $X_i = A_k X''_i$
      $calcFlag = (|r_k X''_i - c_k X_i| > \eta_2)$
    end while
  end for
end procedure

types in this work, Section 3.1 introduces an online scheme aiming at computational faults while Section 3.2 proposes an online scheme that can deal with both computational faults and memory faults. The computational fault tolerant scheme in Section 3.1 is also complementary to ECC memory. It can detect and correct computational errors that ECC may not be able to handle.

3.1 Computational Fault Tolerance

Inspired by the divide-and-conquer nature of FFT algorithm, we leverage this algorithmic characteristic and offline ABFT FFT scheme to propose an online ABFT scheme for FFT. Taking the tradeoff of fault tolerant ability and overhead into consideration, we propose a two-layer ABFT approach that leverages the highest level of decomposition of a Cooley-Tukey FFT to protect the first part and second part by two separate ABFT schemes.

From the view of the highest level of decomposition, an N-point FFT is calculated by computing k m-point FFTs, twiddle multiplications and m k-point FFTs when $N = m * k$. The k m-point FFTs can be protected separately by the ABFT approach. So can the m k-point FFTs. Also, twiddle multiplication can be protected by DMR with low overhead because it is memory-intensive. Thus, the structure of online ABFT scheme can be shown in Fig. 1. The colored parts are protected by their own FFTs while the red parts, including the twiddle multiplication and input checksum vector generation, are protected by DMR.

According to Fig. 1, the input checksum $c_m = r_m A_m$ should be calculated at first. Then the m-point FFTs are executed and verified one by one. If there is error in the i-th FFT, it can be detected by comparing the checksum $c_m x_i$ and $r_m X_i$. It will be corrected by an immediate re-execution of this FFT. Also, the output of the re-calculation would be verified. After that, each element in the intermediate output will multiply itself with the corresponding twiddle factor ($\omega_N^{n_1 j_2}$) to generate the input for the latter k-point FFTs. Then the k-point FFTs are executed and verified one by one. They can be protected by the same mechanism with input checksum $c_k = r_k A_k$. If an error occurs during the execution of any k-point FFT, it can be detected and corrected as the first part. However, if an error strikes the twiddle multiplication, the ABFT scheme cannot detect the error since the input has already been corrupted. Therefore, online DMR is equipped for the twiddle multiplication. Each multiplication is executed twice and verified immediately to ensure correctness. If an error is detected here, a third execution is performed and the final result would be the majority of the three executions. Since computation would only happen in one of the three parts or in the checksum calculation, any single computational error can be revealed. Besides these parts, the other parts are protected by one and only one ABFT FFT so that no computation is wasted. This ensures no masked error and no repeated protection on the same data.

The algorithm of this approach is shown in Algorithm 2. Compared with the offline scheme, the two-layer online scheme only needs to compute two input checksum vectors of size m and k while the offline one needs to compute one input checksum vector of size N. As this computation is one of major overhead, the online scheme should have better performance. Furthermore, since each small FFT is equipped with separate protection, the online scheme is expected to achieve timely recovery when an error occurs.

3.2 Memory Fault Tolerance

Besides the logic units, faults may also strike memory to cause memory errors. This may be even more common than computational errors. If memory fault strikes some intermediate result during computation in some decomposed FFT, this error would behave like a computational error and can be detected and recovered by the ABFT schemes above. However, if it strikes the input before the calculation or the output after the calculation, the error cannot be detected by this scheme alone. Thus, more strict mechanisms are needed to tolerate memory faults.

As usual, two checksums $r_1 = (1, 1, \ldots, 1)$ and $r_2 = (1, 2, \ldots, n)$ are used to detect and recover from a memory error. If any error occurs and changes the input x_j into x'_j, the difference will be:

$$r_1 x - r_1 x' = x_j - x'_j$$

$$r_2 x - r_2 x' = j(x_j - x'_j)$$

Then the error can be located by $(r_2 x - r_2 x')/(r_1 x - r_1 x')$ and recovered by adding $r_1 x - r_1 x'$ to the corrupted element.

In our fault model for the memory faults, we assume that memory faults would not occur when the checksums are being generated, otherwise, the error cannot be detected by ABFT approaches. This is reasonable because the checksum generation would only take very little time (the time complexity is $O(N)$ and its coefficient is very small). Our basic idea to detect memory error is to verify data before use. Denote CCG as computational checksum generation, MCG as memory checksum generation, CCV as computational checksum verification, MCV as memory checksum verification, TM as twiddle multiplication, s as the number of FFTs to be computed together, then the hierarchy of memory protection is shown in Fig. 2. Bold italic operations are original operations in FFTW. To ensure the correctness of the input, memory checksums of each m-point FFT

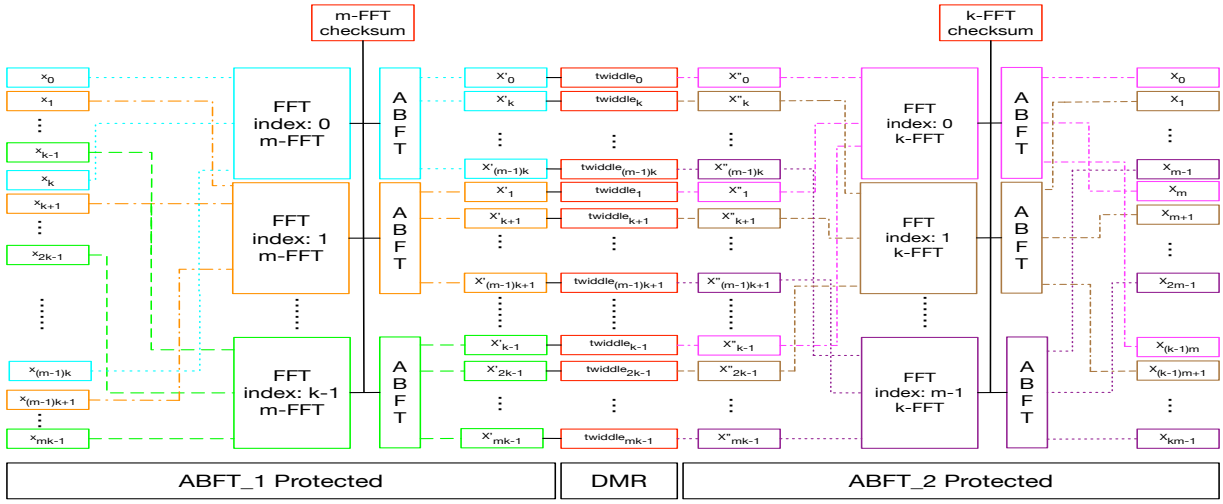

Figure 1: The Two-Layer ABFT FFT Scheme (When $N = m * k$)

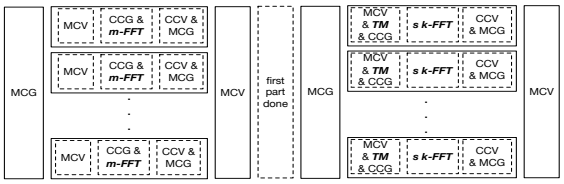

Figure 2: Hierarchy of Memory Protection

are generated before any of the m-point FFT calculations. Then the k m-point FFT calculations would start one by one and verifications are invoked at the beginning of these calculations. If an error occurs, the corrupted input will be located and recovered by the 2 checksums and a restart will be performed immediately. Otherwise, the computation is thought as fault-free and memory checksums for the intermediate output are generated. These checksums will be used for verification before the twiddle multiplication to make sure there is no memory error in the output between the end of this m-point FFT and the end of all the k m-point FFTs.

A similar technique can be applied to the second part. Each k-point FFT needs memory checksum verification before computation, computational checksum verification, and output memory checksum generation after computation. At last, the final output is verified to ensure correctness of the result.

Besides the protection of the input, output and intermediate result, the input checksum vector rA for the m-point FFT and k-point FFT should also be checked. These verifications can be done in time intervals related to the error rate, which is quite feasible across the whole computation. As there is only $O(\sqrt{n})$ time consumed in each verification, it would introduce very little overhead.

This mechanism helps a lot in correcting the memory errors. All the memory errors can be detected and recovered as long as two memory errors do not strike the same FFT at the same time.

4 SEQUENTIAL OPTIMIZATIONS

The implementation of *FFTW* is tricky. It is not easy to add fault tolerance while keeping the same performance. This section introduces some optimizations that we apply to minimize overhead.

4.1 Memory Checksum Modification

Though the traditional memory checksums $r_1 = (1, 1, \ldots, 1)$ and $r_2 = (1, 2, \ldots, n)$ work well for correcting memory error, they may involve redundant computation because they do not make use of the computational checksum $r = (\omega_3^0, \omega_3^1, \ldots, \omega_3^{N-1})$. Since rAx will be calculated under any circumstance to detect computational error, r_1 can be replaced by $r_1' = r$ directly to save the computation time of $r_1 x$. Correspondingly the j-th element in the second checksum r_2 can be replaced by $(r_2')_j = j * (rA)_j$. Similar to the original checksum r_1 and r_2, the difference the new checksums would be:

$$r_1'x - r_1'x' = (rA)_j(x_j - x_j')$$

$$r_2'x - r_2'x' = j * (rA)_j(x_j - x_j')$$

Then the error can be located by $(r_2'x - r_2'x')/(r_1'x - r_1'x')$. After that correction can be done by adding $(r_1'x - r_1'x')/(rA)_j$ to the corrupted element. As the generation time for r_1' and r_2' is $O(\sqrt{N})$, the extra overhead on input checksum vector generation would be negligible. On the other hand, it saves the checksum generation time since it only costs $10N$ operations ($8N$ for $r_1'x$, $2N$ for $r_2'x$) while the original one costs $14N$ ($8N$ for rx, $2N$ for r_1x, $4N$ for r_2x).

4.2 Verification & Correction Postponing

According to Fig. 2, there is input memory checksum generation when FFT starts, followed immediately by memory checksum verification and m-point FFTs. Inspired by the fact that the errors, both computational errors and memory errors, would propagate to the end of each decomposed FFT, MCVs before each m-point FFT can be postponed to the CCVs after this m-point FFT. Since CCV can detect the error, the postponed MCV is eliminated.

Similarly, the MCVs after k-point FFTs can be postponed to the final MCV and these MCVs as well as the MCGs after k-point FFTs can be eliminated for lower overhead. Unfortunately, this cannot be done directly since the second part is always done in-place where the input will be overwritten by the output. If the output verification is postponed, the error can still be detected since the checksums will not match. However, it cannot be corrected since the input is overwritten. Thus, another copy of the intermediate output is

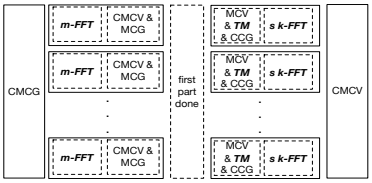

Figure 3: Optimized Hierarchy of Memory Protection

needed. It can be copied to the original input array for no extra memory. Though the copy operation also involves N elements, it would be much faster than the original redundant MCVs and MCGs.

Besides, the correction operations $r_2'x$ can be postponed to the time when an error is detected. However, it will result in slower recovery. When the error rate is low, the optimization can be adopted for lower overhead in error-free runs. When the error rate is high, the correction operations should not be postponed.

4.3 Incremental Checksum Generation

From Section 3.2, the MCG before twiddle multiplication is necessary because there is a rearrangement of data between the two ABFT parts. However, the verification mechanism still seems inefficient since each element is verified twice. Instead of regeneration, this optimization uses incremental generation for the checksums to reorganize the memory checksums.

After the input checksums are generated at the very beginning, extra space is allocated to store the information of the output. Unlike the previous approach, these output checksums directly store the checksums for the k-point FFTs in the second part. At first, these checksums are initialized to 0. At the end of each m-point FFT, the k outputs increase their corresponding slots by their own value, i. e. the first element X_0' would increase the first slot in the checksum by X_0' while the second element X_1' would increase the second slot by X_1'. By this means, the j-th slot in the checksum would happen to be the checksum for elements in the j-th k-point FFT. Thus only one verification is needed before the second part.

4.4 Non-contiguous Memory Access

When a big FFT is broken down into smaller ones, the inputs of each smaller FFT would be non-contiguous as the first k m-point FFTs in Fig. 2. The stride (distance between adjacent inputs) of each m-point FFT would be $2k$. It is usually $O(\sqrt{N})$ and will result in low spatial locality in the cache. Besides basic use in FFT to compute the result, the inputs are also needed in CCGs and MCVs. Another read would be relatively expensive since there would be cache misses all the time, which leads to large overhead. This happens to MCG in the first part. To resolve this, the corresponding MCGs are brought forward to the beginning of all the m-point FFTs and the new MCGs are computed via the incremental checksum generation approach above. It actually accesses each element twice. But each access has little low overhead due to cache reuse.

Denote CMCG as the modified checksum generation and CMCV as the modified checksum verification in Section 4.1, the hierarchy of memory protection can be simplified to Fig. 3 with all the optimizations above. Compared to the original hierarchy in Fig. 2, the optimized one is much simpler and faster.

5 ONLINE ABFT FFT ON PARALLEL SYSTEMS

FFT of large sizes becomes very common nowadays [11]. Therefore, FFT may need to be performed in parallel to avoid the limited memory and low computational efficiency on single processor when FFT size becomes large. Although the idea of sequential ABFT FFT can be borrowed, challenge comes that parallel FFTs are always done in-place for better utilization of memory. In-place and out-of-place are property of an algorithm. In-place means that the algorithm will be done without auxiliary data structure. To make it simple for FFT, the in-place algorithm will store the output in the original input memory and does not bother to allocate a new memory space of size N. The out-of-place algorithm will allocate the memory space to store output in the beginning of the algorithm.

To compute parallel FFT, *FFTW* tends to choose a plan which computes $\frac{N}{p}$ p-point FFTs at first and then p $\frac{N}{p}$-point FFTs. Unfortunately, the data needed for each FFT is not always on the same processor. Thus communication among processors is needed during the computation. Assume FFT size is N and the number of processors is p. Data on each processor is divided into p blocks of size $\frac{N}{p^2}$. Then a six-step algorithm that involves 3 transpositions is adopted for 1D parallel FFTs. A transposition is a communication that exchanges the i-th block of data in processor i with the j-th block of data in processor j for all i and j from 0 to $p-1$. Denote the $\frac{N}{p^2}$ p-point FFTs on a processor as FFT_1 and the latter $\frac{N}{p}$-point FFT as FFT_2. The first transposition is performed at first to deliver data needed for FFT_1 to the same processor. Then FFT_1 is done on each processor in parallel. After that, the second transposition occurs to exchange data for FFT_2. FFT_2 is performed as the next step. When FFT_2 is done, the third transposition is executed to deliver data to its belonging processor. At last, there is some local adjustment to place the final output in a correct order.

Because original input will be overwritten by output, the restart would not work for in-place FFTs. Fig. 4 shows the flowchart of adding fault tolerance to in-place FFTs. Compared to the out-of-place protection in the sequential scheme, input in each in-place FFT should have a backup in case an error occurs. Also, checksum verifications should be done immediately after the output is generated. When a memory error is detected, it should be corrected right away. After that, the input will be recovered by the backup and a restart will be performed.

FFT_1 can be protected by the mechanism above because each p-point FFT only asks for $2p$ space. However, FFT_2 cannot be protected in this way because space will be doubled. Fortunately, the idea of the online sequential ABFT scheme can be applied here for timely detection, faster recovery and less space overhead because FFT_2 will be decomposed to smaller FFTs. Nevertheless, the sequential ABFT scheme cannot be leveraged directly because in-place FFTs tend to select a different execution plan from out-of-place FFTs for efficiency. For example, if $\frac{N}{p}$ is a square number, *FFTW* may choose a plan similar to the out-of-place one to employ a two-layer decomposition; if it is not, i. e. $\frac{N}{p} = r * k^2$, *FFTW* would prefer a more complicated plan. It may perform $r * k$ k-point FFTs at first, then do twiddle multiplications and k^2 r-point FFTs, finally another twiddle multiplications and $r * k$ k-point FFTs. In this situation, the original two-layer online ABFT can no longer

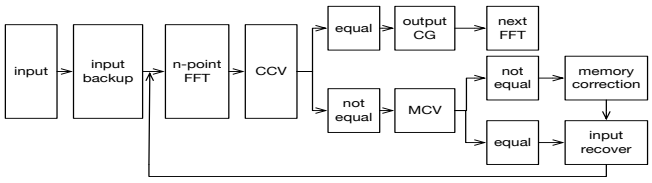

Figure 4: Flowchart of Protected In-Place FFT.

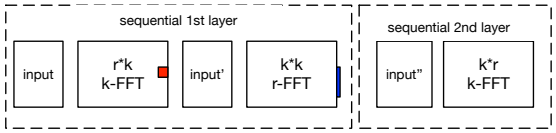

Figure 5: Sequential ABFT scheme no longer works: if an error occurs in the red part, it will be detected in the blue part. At this time, the procedure has to fail since original input is overwritten. Twiddle multiplication is omitted.

work as shown in Fig. 5. Because the FFT is done in-place, the initial input is overwritten after the $k*r$ k-point FFTs so any restart after the $r*k$ k-point FFTs cannot be performed. A checkpoint for input would definitely work here. However, it will have 100% space overhead and longer correction time.

The solution to this kind of plan is to add one flexible verification layer between the original two layers. The added layer would be protected by DMR since r is usually small (2 or 8 for $\frac{N}{p}$ is a power of 2), making the k m-point FFTs an ABFT-DMR scheme. As the execution time of the DMR part is very small (the same magnitude of the time for checksum generation and verification), we can assume there is no memory error in this part. Then the input verification can be brought forward to all the DMR computations and the output checksum generation can be postponed to end of this part.

Besides the modifications on fault-tolerant mechanisms, there are some modifications on communication as well. In order to detect and correct errors that occur in communication, checksums for communicated data should be generated and sent. As there are only 2 checksums for each block of communicated data, the communication overhead would be negligible.

This scheme can be optimized by some of the optimizations mentioned in previous part. After these optimizations, it is good from the sequential point of view because there are no redundant checksum generations and verifications.

6 PARALLEL OPTIMIZATIONS

Besides sequential optimizations that mentioned in previous part, we also adopt several optimizations specifically for parallel FFTs. Some of the optimizations can also be used in fault-free FFTs for better performance. These optimizations are incorporated into our implementation to reduce overhead.

6.1 Computation-Communication Overlap

In FFTW, blocking communication is used for transpositions. It is good because the following step usually needs data from all processors so the non-blocking method would have little benefit. However, the checksum generation and verification in the ABFT FFT scheme

Algorithm 3 Communication-Computation Overlap

```
1:  procedure Non-blocking Transpose
2:      sched[0 to p-1]: schedule for communication
3:      alloc send buffers sb1, sb2 and receive buffers rb1, rb2
4:      generate data for processor sched[0] in sb1
5:      Isend(sb1) to and Irecv(rb1) from processor sched[0]
6:      generate data for processor sched[1] in sb2
7:      Iwait() for processor sched[0]
8:      for i from 1 to p-3 do
9:          Isend(sb2) to and Irecv(rb2) from processor sched[i]
10:         verify and process data from processor sched[i-1] in rb1
11:         generate data for processor sched[i+1] in sb1
12:         Iwait() for processor sched[i]
13:         Isend(sb1) to and Irecv(rb1) from processor sched[i+1]
14:         verify and process data from processor sched[i] in rb2
15:         generate data for processor sched[i+2] in sb2
16:         Iwait() for processor sched[i+1]
17:         increase i by 2
18:     end for
19:     Isend(sb2) to and Irecv(rb2) from processor sched[p-1]
20:     verify and process data from processor sched[p-2] in rb1
21:     Iwait() for processor sched[p-1]
22:     verify and process data from processor sched[p-1] in rb2
23: end procedure
```

are totally uncorrelated with FFT computation, showing great potential for computation-communication overlap.

Our idea for communication-computation overlap is very similar to the idea of pipeline. It doubles the number of send buffer and receive buffer. When $Isend()$ is used to send data in send buffer sb_1 and $Irecv()$ is used to receive data in receive buffer rb_1, data received in another receive buffer rb_2 can be processed and data to be sent in another send buffer sb_2 can be generated. When these operations are done, $Iwait()$ can be used to wait for communication. After that, data in rb_1 can be processed and data to be sent to next processor can be generated in sb_1 while sending data in sb_2 and receiving data in rb_2. The algorithm is shown in Algorithm 3.

With this technique, MCV and CCG before the p-point FFTs can be overlapped with $transpose_1$. MCV, TM and CMCG before the k-point FFTs can be overlapped with $transpose_2$. Besides, the send buffer initialization and receive buffer data transfer in each communication can also be overlapped.

The online ABFT scheme for parallel in-place FFT after overlap is shown in Fig. 6. Bold italic operations are original operations in FFTW. This overlap is optimal since all the other operations are either in the critical path or dependent on the communication. Also, this optimization can be applied to $FFTW$ to overlap the twiddle multiplication in $FFT_{2.1}$ with communication.

6.2 Re-design Plan in FFT_1

Since the input and output are both non-contiguous with a large stride in FFT_1, there is high latency in accessing these elements due to cache misses. Fault-free FFTs do not suffer much from this because the input and output are read and written once during the whole computation. However, with the fault tolerant operations,

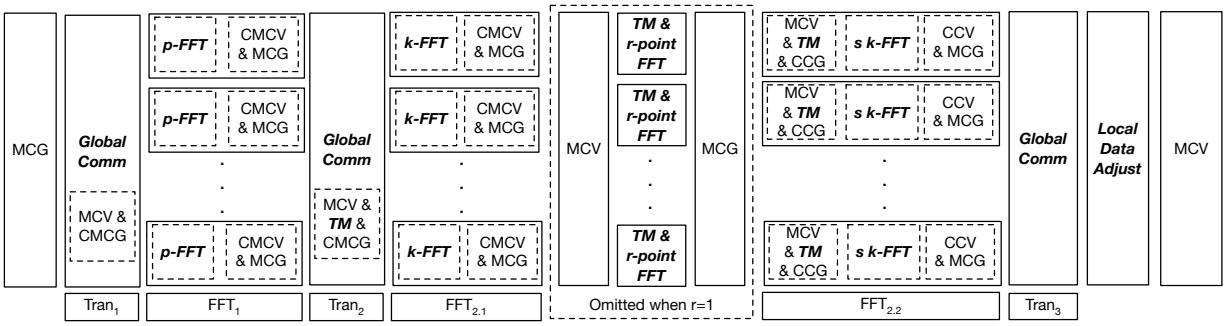

Figure 6: Online ABFT Scheme for Parallel In-Place FFT After Communication-Computation Overlap.

the input and output are at least accessed twice, which may lead to high overhead. Inspired by the implementation of sequential FFT, we use a similar idea to adjust the execution plan of FFT_1.

In order to mitigate the overhead in multiple accesses, one buffer is allocated to store the input contiguously. The input is read into the buffer and computed in the buffer. The result is then verified in the buffer and copied to the output location when the computation is correct. To maximum reuse for data in the cache, the buffer can be made c times the size of data in the p-point FFT, where c is the number of data in the cache line. Each time one element is read into the buffer, the latter c elements are also read and stored in the buffer as well. In this way, cache can be better utilized.

This change may have more operations because there are data assignments between input, output and the buffer. However, it may perform quite well when the p-point FFT barely fits in the cache. In this case, the original implementation would suffer a lot since there is no reuse of cached data. On the other hand, this optimization can make use of cache because data are moved into the buffer. It would be more scalable compared to the original plan.

7 OVERHEAD ANALYSIS

This section analyzes the theoretical overhead for the various schemes above. In the following subsections, c_1, c_2, r_1, r_2 will be used to denote one operation of complex number multiplication, complex number addition, real number multiplication, real number addition. Assume one real number addition or one real number multiplication is the unit of operation, $c_1 = 6$, $c_2 = 2$ and $8r_1 + 3r_2 = 11$ for the complex number division can be derived. This part only discusses the number of operations needed to add fault tolerance. The true overhead may differ since it heavily depends on the implementation. As a comparison, the total number of computational operations in the original FFT would roughly be $5N \log_2 N$.

7.1 Overhead in the Sequential Scheme

7.1.1 Computational FT in the Offline Scheme. The overhead in the offline scheme comes from input checksum vector generation, CCG and CCV. In the offline scheme, rA can be calculated according to characteristics of arithmetic arrays to reduce overhead:

$$(rA)_j = \omega_3^0 \omega_n^{0j} + \omega_3^1 \omega_n^{1j} + \ldots + \omega_3^{n-1} \omega_n^{(n-1)j} = \frac{1 - \omega_3^n}{1 - \omega_3^1 \omega_n^j}$$

Then it can be optimized by replacing trigonometric functions with 2 complex number multiplications. Then the overhead would be:

$$T_{rAGen} = (c_1 + c_1 + 2c_2 + 8r_1 + 3r_2) * N = 27N$$

CCG involves 1 complex number multiplication and 1 complex number addition for each element. Its overhead would be:

$$T_{CCG} = N * c_1 + N * c_2 = 8N$$

As for CCV, the total number of complex multiplications can be reduced to 2 by merging elements of same factors. So the overhead turns out to be:

$$T_{CCV} = 2 * c_1 + N * c_2 \approx 2N$$

Therefore, the total overhead for the offline scheme would be $37N$. If an error occurs, the correction would be another run of the whole FFT and final verification. So the correction time would be $39N + 5N \log_2 N$.

7.1.2 Computational FT in the Online ABFT Scheme. The overhead for the online scheme comes from checksum operations in the two ABFT parts and DMR for input checksum vector generation and twiddle multiplication. DMR for input checksum vector generation is negligible since the checksum sizes are $O(\sqrt{N})$. DMR for twiddle multiplication would cost $12N$ because it needs 2 complex number multiplications.

Overhead for ABFT comes from CCG and CCV. They cost $8N$ and $2N$ respectively. The two ABFT parts have the same overhead. Thus the total overhead for the two-layer ABFT scheme would be:

$$T_{ABFT} = 12N + 2 * (8N + 2N) = 32N$$

If an error occurs in DMR, it will be detected and corrected in no time. If an error strikes the ABFT parts, it will be detected by the ABFT scheme and an FFT of size k or m will be performed. As k and m are usually $\theta(\sqrt{N})$, the recalculation will always be an FFT of size $\theta(\sqrt{N})$, which is negligible. Therefore, the overhead for the online scheme would still be $32N$ even if an error occurs.

7.1.3 Total Overhead in the Offline Scheme. The extra operations in the offline scheme would be the computation of $r_2' x$ when the corresponding optimizations are applied. This computation will cost $4N$ operations. Therefore, the total overhead for the offline scheme would be:

$$T_{offline_m} = 37N + 4N = 41N$$

If there is error, whole computation after checksums generation will be restarted, including the verification operations. The overhead would be $5N \log_2 N + 43N$.

7.1.4 Total Overhead in the Online Scheme. In CMCG, there are $4N$ extra operations for the new checksum r_2' calculation. Besides, there is one more MCG and MCV, which corresponds to $6N$ operations. Also, there is one more CMCV of $2N$ operations in the end. Then the total overhead will be:

$$T_{ABFT_m} = T_{ABFT} + 4N + 6N + 2N + 2N = 46N$$

As the recovery time for both computational error and memory error is negligible, the overhead would still be $46N$ when an error occurs during the execution.

7.2 Sequential Space Overhead

When FFT is calculated on single processor, the space overhead only comes from the checksums of each small FFTs and protection for the buffered intermediate output. As these sizes are at most $4k$ or $4m$, the whole scheme only requires $O(\sqrt{N})$ extra space.

7.3 Overhead in the Parallel Scheme

7.3.1 Overhead Before Communication-Computation Overlap. Before overlap, the fault tolerant operations for parallel online scheme include MCG before $transpose_1$, MCV, CMCG after $transpose_1$, CMCV and MCG before $transpose_2$, MCV, and CMCG after $transpose_2$, CMCV and MCG in $FFT_{2.1}$, 2 MCVs, CCG, MCG in $FFT_{2.2}$ and MCV after $transpose_3$. So there are 2 CMCGs, 2 CMCVs, 4 MCGs, 4 MCVs, 1 CCG and 1 CCV when $r = 1$. The overhead for this situation would be:

$$T_{ABFT_{p^1}} = 2 * (12n + 2n + 8n + 2n) + 4 * (6n + 2n) = 96n$$

When $r \neq 1$, there is 1 more MCV and 1 more MCV as well as DMR for TM and r-point FFTs, thus the overhead in this situation is:

$$T_{ABFT_{p^2}} = 96n + 6n + 2n + 12n + 5n \log_2 r = 116n + 5n \log_2 r$$

7.3.2 Overhead After Communication-Computation Overlap. The overlapped communication includes 2 MCVs, 2 CMCGs and 1 TM, thus the new overhead when $r = 1$ would be:

$$T'_{ABFT_{p^1}} = 96n - (2 * (12n + 2n) + 12n) = 56n$$

Similarly, the new overhead when $r \neq 1$ would be:

$$T'_{ABFT_{p^2}} = 116n + 5n \log_2 r - (2 * (12n + 2n) + 12n) = 76n + 5n \log_2 r$$

The correction time for the parallel online scheme would also be negligible since correction in each part would cost negligible time.

7.4 Parallel Space Overhead

Assume the size of used space is $n = \frac{N}{p}$ on each processor, the largest allocated extra memory would be the checksum arrays in FFT_1, which totally take up $\frac{2n}{p}$ space. Besides, there are buffers for communication. Our communication-computation overlap operations allocate four buffers, each of which takes up $\frac{n}{p}$ space. Thus total space overhead would be $\frac{6n}{p}$. The other extra memory are all $O(m)$ or $O(k)$, which is $\theta(\sqrt{n})$. Also, the operations in communication can reuse the space freed from the send and receive buffers in the communication, which requires no extra memory. Therefore, the required extra space would be $\frac{6n}{p}$, then the relative space overhead would be $\frac{6}{p}$.

7.5 Parallel Communication Overhead

The communication overhead of the ABFT scheme comes from the increased message size in the communication. During each communication, the proposed scheme needs to send and receive two checksums for each block of data, which corresponds to an overhead of $\frac{2p^2}{N}$. As there is no extra overhead in the number of messages, the communication overhead would be at most $\frac{2p^2}{N} = \frac{2p}{n}$.

8 IMPACT OF ROUND-OFF ERRORS

Due to the finite word length in floating number arithmetic, round-off errors are unavoidable in software level implementations. Therefore, the two checksums in the ABFT scheme may not be equal even though the whole FFT system is fault free. To avoid the situation above to be diagnosed as faulty, a small difference η between the result is allowed as in previous work. The selection of η is essential because it is a tradeoff between throughput (true negative, fault-free while diagnosed as faulty) and fault coverage (false positive, faulty while diagnosed as fault-free). This section analyzes the estimation of round-off errors and how to choose suitable η.

8.1 Round-off Errors in Computational FT

In existing work, the hardware implementations always employ the fixed-point round off strategy, which is quite different from the floating point arithmetic in the software level. Fortunately, Liu [23], Weinstein [36] and Gentleman [19] have already conducted some research on this topic. Assuming the N real numbers and N imaginary numbers in the input are mutually uncorrelated random variables with zero means. According to [36], the noise-to-signal ratio in an N-point FFT computation would be:

$$\frac{\sigma_E^2}{\sigma_X^2} = 2\sigma_\epsilon^2 \log_2 N$$

Where σ_E^2 is the variance of round-off error, σ_X^2 is the variance of the output, σ_ϵ is the error due to rounding floating point multiplication or addition. σ_ϵ can be assumed uniformly distributed in $(-2^{-t}, 2^{-t})$ or experimentally measured as ${\sigma_\epsilon}^2 = (0.21)2^{-2t}$ in [19], where t is the number of bits in the mantissa part of a floating point.

Assume the input x of an m-point FFT has zero means and variance σ_0. Its output X will have zero means and variance $\sigma_1 = \sqrt{m}\sigma_0$. According to equation (3), the variance of round-off would be $\sigma_e = \sqrt{2m\sigma_0^2\sigma_\epsilon^2 \log_2 m}$. After the summation, the variance of the round off error in the final sum would vary from $\log_2 m * \sigma_e$ to $m * \sigma_e$. To improve fault coverage, we use the upper-bound $m * \sigma_e$ for this estimation. As the input precision loss would be much smaller than the output precision loss, the variance of the final difference would be $\sigma_{roe} = m * \sigma_e = m\sqrt{2m\sigma_0^2\sigma_\epsilon^2 \log_2 m}$. In the k-point FFTs, the input has variance $\sqrt{m}\sigma_0$ and output has variance $\sqrt{km}\sigma_0$. Similarly, we can derive $\sigma_{roe2} = k\sqrt{2km\sigma_0^2\sigma_\epsilon^2 \log_2 k}$.

After that, an approach similar to [35] can be employed to set the coefficient η. According to central limit theory, the throughput

of an N-point FFT can be estimated as:

$$throughput(\eta, N, \sigma_i) = \frac{1}{1 + P(\frac{|F|}{\sqrt{N}\sigma} > \frac{\eta}{\sqrt{N}\sigma})} = \frac{1}{3 - 2\Phi(\frac{\eta}{\sqrt{N}\sigma})}$$

When $\eta = 3\sqrt{N}\sigma$, the theoretical throughput is 0.997. According to this formula, different η can be set to different parts of the online ABFT scheme. I.e., $\eta_1 = 3\sqrt{m}\sigma_{roe}$, $\eta_2 = 3\sqrt{k}\sigma_{roe2}$ can be chosen respectively for m-point FFTs and k-point FFTs in sequential FFT. In parallel FFT, things are similar. The only difference is that there are three ηs to be set respectively for FFT_1, $FFT_{2.1}$ and $FFT_{2.2}$.

8.2 Round-off Errors in Memory FT

Memory round-off errors would be much smaller since it only involves simple summation. According to the analysis above, the summation of m elements in the array x will result in a variance $m * \sqrt{var(x)}\sigma_\epsilon$ in the precision loss in the result for data with high precision. Then threshold can be set by the approach above.

9 EXPERIMENTAL EVALUATIONS

We implement the proposed ABFT scheme into the widely used FFTW library [15, 16] - one of the fastest software implementations of FFT and reports the experimental results in this section.

9.1 Experiment Setup

We evaluated our implementation on TIANHE-2, the current *2nd* fastest supercomputer in the world. Each node of TIANHE-2 has 2 E5-2692 processors (with *24 cores* in all) and 64GB memory.

9.2 Overhead in Sequential Scheme

This section evaluates the sequential schemes for out-of-place FFT on single processor. FFT sizes from 2^{25} to 2^{28} are tested. Each experiment is run 9 times and the average number is recorded.

9.2.1 Experiments without Fault. Four schemes are evaluated at this part and the results are shown in Fig. 7. Fig. 7(a) shows the evaluations for computational FT schemes. The first bar shows the overhead of the naive offline scheme. The second bar is the evaluation of the optimized offline scheme. A naive online scheme is displayed as the third bar and an optimized online scheme is shown as the last bar. Fig. 7(b) shows the evaluations for computational and memory FT schemes. The only difference is that the third bar displays the online scheme with computational FT optimizations.

From the figure, we can see that the optimization techniques play an important role in the FT-FFT schemes. The optimized offline scheme is much better than the naive offline scheme due to the number of calls to the trigonometric functions. The optimized online scheme outperforms the offline one a lot when only computational errors are considered. Also, it has comparable performance to the optimized offline scheme even when memory errors are considered.

9.2.2 Experiments with Faults. This part shows the timely recovery of the online scheme. As the offline scheme only guarantees to detect one error, only one memory fault is injected in the optimized offline scheme. Three fault injections are performed on the online scheme: one computational fault ($1c$); one computational fault and a memory fault ($1m + 1c$); two computational faults and one memory fault ($1m + 2c$). (0) indicates fault-free executions as

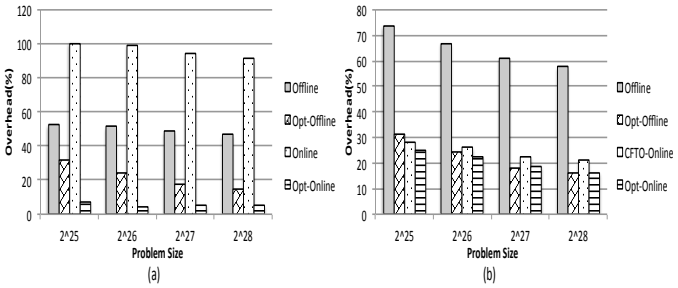

Figure 7: Overhead of ABFT-FFT Schemes on TIANHE-2 When There Is No Fault: (a) Computational FT (b) Computational & Memory FT

Table 1: Execution Time (Seconds) Comparison of FT-FFT on TIANHE-2 When There Are Faults

Problem Size	$N = 2^{25}$	$N = 2^{26}$	$N = 2^{27}$	$N = 2^{28}$
$FFTW(0)$	3.71	8.04	16.79	34.97
$Opt-Offline(0)$	4.88	10.01	19.86	40.52
$Opt-Offline(1m)$	9.63	20.21	42.89	87.65
$Opt-Online(0)$	4.64	9.83	19.94	40.64
$Opt-Online(1c)$	4.78	9.92	20.17	40.92
$Opt-Online(1m+1c)$	4.83	9.98	20.44	41.28
$Opt-Online(1m+2c)$	4.86	10.17	20.77	41.68

comparison. Computational fault is simulated as adding some constant to an element while memory fault is simulated by changing one element to another constant. Table 1 shows the execution time of the optimized schemes with different number of errors.

According to the table, the online scheme does have strong fault tolerant ability. The offline scheme suffers from the re-execution when an error occurs thus it costs about twice the time the online scheme does. On the other hand, because one error only leads to a recalculation of a m-point FFT or s k-point FFTs which costs $O(\sqrt{N}\log\sqrt{N})$ time, the execution time of the online scheme can almost maintain the same when the number of errors increases. In fact, as long as no two errors strike the same m-point FFT or s k-point FFTs at the same time, the online scheme is able to detect and correct all of them quickly. Therefore, the online scheme is able to perform well even when the error rate is relatively high, showing great advantage over the offine scheme.

9.3 Performance in Parallel Scheme

This section evaluates the parallel online scheme for in-place FFT in large scale. Because of fluctuations, each experiment is run 20 times and the average number is recorded.

9.3.1 Experiments without Fault. Three implementations together with original $FFTW$ are evaluated at this part. The results of weak scaling and strong scaling are shown in Fig. 8. The first bar shows the execution time of original $FFTW$. The sequentially optimized fault tolerant scheme $FT-FFTW$ is displayed as the second bar. The third bar $opt-FFTW$ is $FFTW$ with parallel optimizations in Section 6. The last bar $opt-FT-FFTW$ is the parallel fault tolerant scheme with both sequential and parallel optimizations. According to the figure, the sequentially optimized ABFT scheme has some overhead over the original $FFTW$. The overhead comes from the checksum operations. On the other hand, the online scheme with

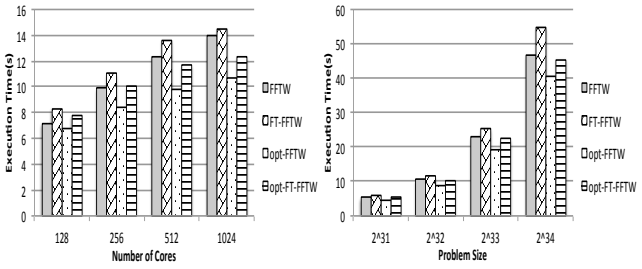

Figure 8: Execution Time (Seconds) of Parallel FT-FFT Schemes on TIANHE-2 When There Is No Fault: (a) Strong Scaling: $n = 2^{26}$ (b) Weak Scaling: $p = 256$

Table 2: Execution Time (Seconds) Comparison of Strong Scaling FT-FFTW on TIANHE-2 When There Are Faults

Number of Cores	$p = 128$	$p = 256$	$p = 512$	$p = 1024$
$Opt-FT-FFTW(0)$	7.83	10.24	11.34	12.47
$Opt-FT-FFTW(2m)$	7.85	10.23	11.39	12.57
$Opt-FT-FFTW(2c)$	7.85	10.28	11.33	12.59
$Opt-FT-FFTW(2m+2c)$	7.86	10.23	11.34	12.56

Table 3: Execution Time (Seconds) Comparison of Weak Scaling FT-FFTW on TIANHE-2 When There Are Faults

Problem Size	$N = 2^{31}$	$N = 2^{32}$	$N = 2^{33}$	$N = 2^{34}$
$Opt-FT-FFTW(0)$	5.45	10.35	22.45	45.63
$Opt-FT-FFTW(2m)$	5.42	10.35	22.55	45.31
$Opt-FT-FFTW(2c)$	5.43	10.36	22.47	45.46
$Opt-FT-FFTW(2m+2c)$	5.45	10.31	22.55	45.47

parallel optimizations beats the original $FFTW$ in error-free runs because the parallel optimizations work very well. However, it still has some overhead over $opt-FFTW$ due to checksum operations.

9.3.2 Experiments with Faults. This part shows the fault tolerant ability of the parallel online scheme. Fault injection is similar to the one in Section 9.2.2 except that faults are injected in each processor. Experiments of no faults (0), 2 memory faults (2m), 2 computational faults (2c), 2 memory faults and 2 computational faults (2m+2c) are shown in Table 2 and Table 3.

According to the tables, this scheme does have strong fault tolerant ability. It only takes very little time to recover from multiple faults because each fault only revokes a restart of one or several p-point FFTs or $\sqrt{\frac{n}{p}}$-point FFTs. Note that sometimes the error free run may have longer execution time. This is caused by fluctuation.

9.4 Round-off Errors

As parallel FFTs have similar round-off error impact to sequential scheme, only experiments on sequential schemes of 2^{25}-point FFT are tested. These results can be generalized to the parallel scheme.

9.4.1 Round-off Error Approximation. In this part, the accuracy of round-off analysis in Section 8 is evaluated. Input from uniform distribution $U(-1,1)$ and normal distribution $N(0,1)$ is tested respectively. 1000 runs are performed thus there are 8192000 m-point FFTs and 1024000 s k-point FFTs. The result is shown in Table 4.

In Table 4, the column Max_1 shows the max round-off error in the m-point FFTs. Est_1 shows the estimated η for this part. $Thput_1$ shows the throughput of the scheme. The latter three columns show the same property of the k-point FFTs. The selected η provides nearly 100% throughput while keeping close to the round-off error bound. It promises good coverage.

Table 4: Approximation of Round-off Error

$Input$	Max_1	Est_1	$Thput_1$	Max_2	Est_2	$Thput_2$
$U(-1,1)$	$0.92*10^{-8}$	$1.45*10^{-8}$	100%	$0.61*10^{-6}$	$3.86*10^{-6}$	100%
$N(0,1)$	$3.8*10^{-8}$	$2.51*10^{-8}$	99.96%	$1.11*10^{-6}$	$6.69*10^{-6}$	100%

Table 5: Minimal Magnitude of Error That Can Be Detected

$Schemes$	e_1	e_2	e_3
$Offline$	10^{-2}	10^{-2}	10^{-2}
$Online$	10^{-7}	10^{-6}	10^{-6}

Table 6: Distribution of Relative Errors of FFT Output in 1000 Runs When One Random Fault Is Injected in Each Run

$\frac{\|\|x'-x\|\|_\infty}{\|\|x\|\|_\infty}$	$Uncorrected$	$> 10^{-6}$	$> 10^{-8}$	$> 10^{-10}$	$> 10^{-12}$
$No\_Correction$	–	73.4%	82.4%	84.0%	84.2%
$Offline$	4.4%	5.2%	20.8%	33.4%	35.7%
$Online$	2.5%	2.5%	2.5%	2.5%	3.9%

9.4.2 Detection Ability Comparison. This section compares the detection ability of the online scheme and the offline scheme. Same fault is injected into the same position of the different schemes. Three fault injection positions are tested in this part. e_1 is injected in the input after checksum verification; e_2 is injected in the input of the second FFT; e_3 is injected in the final output. In the fault injection, the selected element will increase itself by the given error magnitude. I. e., if the magnitude of error is 10^{-3}, 10^{-3} is added to the selected element and whether the error is detected is observed. η of the offline scheme is set as the round-off error bound of error-free runs to allow for 100% throughput.

From Table 5, the online scheme can detect a much smaller magnitude of errors than the offline scheme. Thus, when throughput is similar, the online scheme should have much larger fault coverage.

9.4.3 Fault Coverage Tests. This section shows the relative errors of FFT output after an error occurs in a 2^{25}-point sequential FFT with input drawn from $U(-1,1)$. As random computational errors are hard to simulate and some of them can be simulated as memory errors, only memory error of single bit flip is tested here.

Some fault-free runs of 2^{25}-point FFT are performed at first to get a rough upper bound of the round-off errors of the offline schemes. After that, η is set as this rough upper-bound to allow for nearly 100% throughput and relative errors are evaluated after randomly flipping one higher bit (flipping lower bit is usually masked) in the input or output array. Define the relative error as $\frac{||x'-x||_\infty}{||x||_\infty}$, where x is the correct output, x' is the output with fault injection and $||\bullet||_\infty$ is the infinity norm of vector. 1000 independent runs are performed and the distribution of relative errors is shown in Table 6. The first row shows the relative error of runs without correction. It indicates the impact of errors on output as a comparison. The second column $Uncorrected$ shows the percentage of uncorrected errors due to wrong indexing caused by round-off errors. It can be improved by changing the indexing checksum r_2. For these situations, the relative error is set as infinite.

According to the table, the online scheme outperforms the offline scheme a lot in fault coverage because the relative errors it introduces are of much smaller magnitude. For example, if the error bound is set as 10^{-12}, the fault coverage in the online scheme would be 96.1% compared to 64.3% in the offline scheme. It shows great potential in practical use.

10 CONCLUSION

This paper presents an online ABFT scheme to correct soft errors online in the widely used FFT computations. The proposed scheme only needs to repeat a small fraction of the computation after errors occur. Experimental results demonstrate that the proposed scheme improves the computing efficiency by 2X over existing schemes when errors occur.

ACKNOWLEDGMENTS

This work is partially supported by the NSF grants OAC-1305624, CCF-1513201, the SZSTI basic research program JCYJ2015063011494-2313, and the MOST key project 2017YFB0202100.

REFERENCES

[1] Anna Antola, R Negrini, MG Sami, and Nello Scarabottolo. 1992. Fault tolerance in FFT arrays: time redundancy approaches. *Journal of VLSI signal processing systems for signal, image and video technology* 4, 4 (1992), 295–316.

[2] Rizwan A Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the propagation of transient errors in HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 72.

[3] Prithviraj Banerjee, Joe T Rahmeh, Craig Stunkel, VS Nair, Kaushik Roy, Vijay Balasubramanian, Jacob Abraham, and others. 1990. Algorithm-based fault tolerance on a hypercube multiprocessor. *Computers, IEEE Transactions on* 39, 9 (1990), 1132–1145.

[4] Patrick G Bridges, Kurt B Ferreira, Michael A Heroux, and Mark Hoemmen. 2012. Fault-tolerant linear solvers via selective reliability. *arXiv preprint arXiv:1206.1390* (2012).

[5] Greg Bronevetsky and Bronis de Supinski. 2008. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 155–164.

[6] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Mark Snir. 2009. Toward exascale resilience. *International Journal of High Performance Computing Applications* (2009).

[7] Marc Casas, Bronis R de Supinski, Greg Bronevetsky, and Martin Schulz. 2012. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 91–100.

[8] Jieyang Chen, Xin Liang, and Zizhong Chen. 2016. Online Algorithm-Based Fault Tolerance for Cholesky Decomposition on Heterogeneous Systems with GPUs. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 993–1002.

[9] A Chien, P Balaji, P Beckman, N Dun, A Fang, H Fujita, K Iskra, Z Rubenstein, Z Zheng, R Schreiber, and others. 2015. Versioned Distributed Arrays for Resilience in Scientific Applications: Global View Resilience. *Journal of Computational Science* (2015).

[10] Yoon-Hwa Choi and Miroslaw Malek. 1988. A fault-tolerant FFT processor. *Computers, IEEE Transactions on* 37, 5 (1988), 617–621.

[11] Thomas H Cormen and David M Nicol. 1998. Performing out-of-core FFTs on parallel disk systems. *Parallel Comput.* 24, 1 (1998), 5–20.

[12] Teresa Davies and Zizhong Chen. 2013. Correcting soft errors online in LU factorization. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, 167–178.

[13] Sheng Di and Franck Cappello. 2016. Adaptive impact-driven detection of silent data corruption for HPC applications. *IEEE Transactions on Parallel and Distributed Systems* 27, 10 (2016), 2809–2823.

[14] James Elliott, Mark Hoemmen, and Frank Mueller. 2014. Evaluating the Impact of SDC on the GMRES Iterative Solver. In *IPDPS*. 1193–1202.

[15] Matteo Frigo and Steven G Johnson. 1998. FFTW: An adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, Vol. 3. IEEE, 1381–1384.

[16] Matteo Frigo and Steven G Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[17] Hongyi Fu and Xuejun Yang. 2009. Fault tolerant parallel FFT using parallel failure recovery. In *Computational Science and Its Applications, 2009. ICCSA'09. International Conference on*. IEEE, 257–261.

[18] A Geist. 2016. How to kill a supercomputer: Dirty power, cosmic rays, and bad solder. *IEEE Spectrum* (2016).

[19] W Morven Gentleman and Gordon Sande. 1966. Fast Fourier Transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*. ACM, 563–578.

[20] Kuang-Hua Huang and Jacob A. Abraham. 1984. Algorithm-Based Fault Tolerance for Matrix Operations. *Computers, IEEE Transactions on* 33, 6 (1984), 518–528.

[21] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Exploiting asynchrony from exact forward recovery for due in iterative solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 53.

[22] Jing-Yang Jou, Jacob Abraham, and others. 1988. Fault-tolerant FFT networks. *Computers, IEEE Transactions on* 37, 5 (1988), 548–561.

[23] Toyohisa Kaneko and Bede Liu. 1970. Accumulation of round-off error in fast Fourier transforms. *Journal of the ACM (JACM)* 17, 4 (1970), 637–654.

[24] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S Vetter. 2013. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 44.

[25] Choong Gun Oh and Hee Yong Youn. 1993. On concurrent error detection, location, and correction of FFT networks. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*. IEEE, 596–605.

[26] Choong Gun Oh, Hee Yong Youn, and Vijay K Raj. 1995. An efficient algorithm-based concurrent error detection for FFT networks. *Computers, IEEE Transactions on* 44, 9 (1995), 1157–1162.

[27] Laercio L Pilla, P Rech, F Silvestri, Christopher Frost, Philippe Olivier Alexandre Navaux, M Sonza Reorda, and Luigi Carro. 2014. Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs. *Nuclear Science, IEEE Transactions on* 61, 4 (2014), 1874–1880.

[28] Piyush Sao and Richard W. Vuduc. 2013. Self-stabilizing iterative solvers. In *ScalA*. 4:1–4:8.

[29] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. 2011. Characterizing the impact of soft errors on iterative methods in scientific computing. In *ICS*. 152–161.

[30] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. 2012. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *ICS*. 69–78.

[31] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. 2012. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *DSN*. 1–12.

[32] Miroslav Stoyanov and Clayton Webster. 2015. Numerical analysis of fixed point algorithms in the presence of hardware faults. *SIAM Journal on Scientific Computing* 37, 5 (2015), C532–C553.

[33] DL Tao and Carlos R. P. Hartmann. 1993. A novel concurrent error detection scheme for FFT networks. *Parallel and Distributed Systems, IEEE Transactions on* 4, 2 (1993), 198–221.

[34] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z Zhang, Darren Kerbyson, and Zizhong Chen. 2016. New-sum: A novel online abft scheme for general iterative methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 43–55.

[35] Sying-Jyan Wang and Niraj K Jha. 1994. Algorithm-based fault tolerance for FFT networks. *Computers, IEEE Transactions on* 43, 7 (1994), 849–854.

[36] C Weinstein. 1969. Roundoff noise in floating point fast Fourier transform computation. *IEEE Transactions on Audio and Electroacoustics* 17, 3 (1969), 209–215.

[37] Panruo Wu and Zizhong Chen. 2014. FT-ScaLAPACK: Correcting soft errors online for ScaLAPACK Cholesky, QR, and LU factorization routines. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 49–60.

[38] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. 2017. Silent Data Corruption Resilient Two-sided Matrix Factorizations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 415–427.

[39] Panruo Wu, Chong Ding, Longxiang Chen, Feng Gao, Teresa Davies, Christer Karlsson, and Zizhong Chen. 2011. Fault tolerant matrix-matrix multiplication: correcting soft errors on-line. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems*. ACM, 25–28.

[40] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. 2016. Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 31–42.