Application-Based Fault Tolerance Techniques for Fully Protecting Sparse Matrix Solvers

Grzegorz Pawelczak
HPC Group
University of Bristol
Bristol, United Kingdom
g.pawelczak@bristol.ac.uk

Simon McIntosh-Smith
HPC Group
University of Bristol
Bristol, United Kingdom
s.mcintosh-smith@bristol.ac.uk

James Price
HPC Group
University of Bristol
Bristol, United Kingdom
j.price@bristol.ac.uk

Matt Martineau
HPC Group
University of Bristol
Bristol, United Kingdom
m.martineau@bristol.ac.uk

***Abstract*—The continuous growth of high-performance computing (HPC) systems has lead to Fault Tolerance (FT) being identified as one of the major challenges for exascale computing, due to the expected decrease in Mean Time Between Failures (MTBF). One source of faults are soft errors, which can cause bit corruptions to the data held in memory. Current solutions for protection against these errors include hardware Error Correcting Codes (ECC), which incur overheads in power, memory bandwidth and storage, while also introducing more complexity to the hardware. In this paper we demonstrate Application-Based Fault Tolerance (ABFT) as an alternative method of protecting sparse matrices and dense vectors from data corruptions, requiring no additional dedicated memory storage. We use TeaLeaf, a heat conduction miniapp from the Mantevo Project, to demonstrate how these ABFT techniques can be adapted and applied to a sparse matrix solver-based application and its underlying data structures in order to improve reliability and performance.**

***Index Terms*—Exascale; Fault Tolerance; Linear Sparse Matrix Solvers; Resilience**

I. Introduction

The original Exascale report by DARPA [1] has outlined resiliency as one of the major four challenges faced by Exascale computing. Part of this challenge is to ensure the system has the ability to detect and handle faults which occur during computation. One major source of such faults are errors occurring in the memory of the system. These have been extensively examined, with multiple studies investigating faults which occur in DRAM and SRAM (e.g. [2], [3], [4], [5], [6]) as well as GPU memory (e.g. [7], [8], [9]).

These studies have identified the following error types that can cause faults in memory:

- Hard errors, which are caused by hardware failure (whether permanent or temporary), for example DRAM device failures or bits stuck at 1 or 0.
- Soft errors, usually caused by cosmic rays [10], which can trigger upsets and flip bits in memory without any permanent damage to the hardware.

Faults in memory, whether they are hard or soft, can cause the following kinds of errors:

- Detectable Correctable Errors (DCE) - these are errors which can be detected and corrected to the original state by the system. These errors are usually not a source of concern unless they keep occurring in the same location.
- Detectable Uncorrectable Errors (DUE) - these errors can be detected by the system, but not corrected. Unless a recovery method is used, such as Checkpoint-Restart, these errors usually cause a system failure.
- Silent Data Corruptions (SDC) - these occur when an error exceeds the error detecting capabilities of the system, therefore going undetected by the system, or potentially causing the system to attempt an erroneous correction.

SDCs can cause serious errors in calculations and so most HPC systems deploy mechanisms to prevent these from occurring, such as Error Correcting Codes (ECC). The aim of ECC is to prevent SDCs by providing sufficient error detecting capabilities, as well as reducing, but not completely eliminating, the number of DUEs. ECC protects the memory of a system by adding additional redundant data, requiring additional storage. This redundant data is then used to determine if any errors in memory have occurred and if possible, these errors are then corrected.

Common hardware implementations for protecting the data stored in memory include Single Error Correction and Double Error Detection (SECDED) Hamming codes [11], and Single Symbol Correct and Double Symbol Detect (SSCDSD) Chipkill [12]. Typical SECDED implementations add 8-bits of redundant data for every 64 bits of data, whereas Chipkill adds 16-bits for every 128 bits of data; both schemes therefore add 12.5% overhead in terms of memory required to store the redundant data. This extra overhead in memory storage means that both of these solutions require extra DRAM devices to be added to the DIMM, which increases the cost of memory, and hence the total cost of the system. These extra devices also increase the energy consumption of the DIMMs by at least 12.5% and with memory access expected to account for about 30% of the 20MW power budget of Exascale systems [13], this overhead can be very costly.

Another downside of Chipkill is that SSCDSD can only use x4 DRAM devices, which are 30% less energy efficient than x8 DRAM devices [14]. The use of x4 DRAM devices also means that each 128-bit wide memory request needs to access 36 DRAM devices, significantly more than when using x8 DRAM devices, further reducing the energy efficiency [15]. Whilst the error rates of individual DRAM devices seem to stay constant between different generations [16], the total number of faults is

likely to increase. This is because at Exascale the total number of DRAM devices is expected to be much higher; for example, the current number 3 HPC System on the Top500, Piz Daint, has around 0.3PB of memory [17], whereas early Exascale machines are predicted to have around 50PB of memory [18], a two orders of magnitude increase.

Hardware ECC, usually SECDED, is also available in many HPC GPUs which store the redundant bits along with the data. However this means that when ECC is enabled, the memory size and bandwidth available are reduced by around 12.5%, with a corresponding increase in energy consumption during memory accesses.

II. Contributions

In this paper we make the following specific contributions regarding the fault tolerance of sparse matrix-based solver methods:

1) We extend prior Application Based Fault Tolerance (ABFT) methods to fully protect the sparse matrix data stored in CSR format and the dense floating point vectors. These methods require no extra storage overhead. We also provide a method to reduce the overheads of the previously proposed techniques by taking advantage of the properties of the Conjugate Gradient (CG) method.
2) We implement our techniques in the TeaLeaf miniapp [19], and use TeaLeaf to provide performance results on different architectures including, for the first time, on GPUs using CUDA.
3) We compare different Error Detecting and/or Correcting Codes to show the possible trade-offs between resilience and runtime performance on multiple platforms.

The rest of this paper is structured as follows. In "Previous Work", we talk about previous latest work which is relevant to our research. In Section "Error Detecting and Correcting Codes" we provide an overview of the ECC methods which we are using to detect and/or correct bit flips in the data of the program. In Section "Sparse Matrix Solvers" we give an overview of TeaLeaf, the miniapp we have used to demonstrate how our techniques can be applied to sparse matrix solvers. We also describe the underlying data structures used by the solvers and highlight how they can be used to provide FT. The following Section "Protecting Sparse Matrix Solvers" details of our efficient software based ECC with no storage overhead. The Section "Performance Results" provides detailed performance results for all of these techniques running on a range of devices which are commonly found in HPC clusters. Finally in "Performance Results" we provide our conclusions and discuss ideas for future work.

III. Previous Work

Previous work by McIntosh-Smith et al. [13] introduced ABFT techniques for protecting sparse matrices stored in either Coordinate (COO) or Compressed Sparse Row (CSR) from SDCs. In this research the unused bits from the index vectors were re-purposed to store the ECC data, meaning no extra storage is required.

This work was then further extended in [20], where the software ECC schemes were ported to TeaLeaf, with the addition of software-based CRC32C methods to protect the Sparse Matrix elements themselves.

The techniques presented in this previous research however do not fully protect the whole CSR matrix, as the row integer vector has been left unprotected. Our new work improves on earlier results, in that all the data structures are now protected, including the dense integer and double floating point vectors. In this new work we also investigate the performance of the proposed techniques on a diverse range of architectures, including both HPC and consumer GPUs.

IV. Error Detecting and Correcting Codes

In order to deal with noise in the data, such as cosmic rays causing bit flips in memory, Error Detecting and/or Correcting Codes have become the standard method to mitigate the problem. These methods add redundant bits to the original data, using an algorithm in order to form a codeword. This redundancy increases the minimum Hamming Distance (HD) between the codewords, which allows the receiver to detect a limited number of errors that may occur anywhere in the program data, and often to correct these errors without a program failure.

One of the simplest and least computationally expensive functions to calculate redundancy is Single Error Detection (SED). This ECC scheme works by calculating the parity of the data, and the parity is then added to the data to form a codeword. When performing integrity checks, the whole codeword is calculated, and if it is a non-zero value then bit flips have occurred. This Error Detecting Code (EDC) provides a minimum HD of 2, which means that it can detect one bit flip (or actually all odd numbers of bit flips). However SED on its own cannot correct any of these errors, and it will completely miss any even number of bit flips.

Another ECC method we investigate in this paper is the SECDED Hamming code, which is able to correct a single bit flip, or detect a double bit flip in a codeword. In our research we consider two versions of SECDED, SECDED64 and SECDED128, providing protection for 64 bits and 128 bits of data respectively. The SECDED64 method requires significantly more redundant bits to be stored compared to SECDED128, as the 64-bit version adds 8-bits of redundancy per 64-bits of data, whereas the 128-bit version adds 9-bits of redundancy per 128-bits of the data. This difference however means that SECDED128 can only correct or detect half as many errors as SECDED64 can per 128-bits of data.

When performing SECDED integrity checks, a syndrome vector and parity of the codeword are calculated. If the parity of the codeword is a non-zero value, then a single error correction is performed using the syndrome to uniquely identify the location of the bit flip inside the codeword. If the parity is zero, but the syndrome vector is a non-zero value then this indicates that a double bit error has occurred, but it cannot be corrected and so another method such as Checkpoint-Restart would be required to recover from this error. If three or

more bit flips occur within the same codeword, then SECDED might in the best case detect these, but in the worst case it may cause SDCs by either attempting to perform an erroneous correction of the codeword, or by not being able to detect any of the bit flips at all.

The final ECC code that we use to provide protection against bit flips is a Cyclic Redundancy Check (CRC) code. This checksum-based code treats the data as a polynomial in the Galois field of two elements. This polynomial is then divided by $G(x)$, which is a predefined generator polynomial; the remainder of this polynomial division is used as the redundancy data. When an integrity check is performed, the CRC checksum is removed from the codeword, the CRC value is recalculated for the data and compared with the previously stored value. If any bit flips have occurred then there will be a difference between these checksums, which can then be used to determine the location of bit flip(s).

In this work we focus on CRC32C, a particular type of CRC which adds 32-bits of redundant data per codeword. This CRC type has a generator polynomial with a $(x + 1)$ factor, which means that it can detect all odd bit errors and also *burst errors*[1] of length up to and including 32 bits long [21]. Although CRC is often considered an EDC, its error correcting capabilities are often overlooked. The ability to correct bit flips by CRC is not an easy thing to determine, because the minimum HD depends on multiple factors, including the generator polynomial and the length of the data; determining this minimum HD is an NP problem. However given that we know what generator polynomial we are using and we also know the codeword sizes, these values can be pre-calculated. In our case when using CRC32C, if we choose codewords of size in the range 178 to 5,243 bits, then the minimum HD is 6 [21], meaning that CRC can be used to protect for up to 5 bit errors within each codeword. More specifically, if we consider a code that can correct n bit errors and detect m bit errors, denoted nECmED, then we can form 2EC3ED, 1EC4ED or even 5ED codes ($n + m = 5$). These error correcting capabilities do not have a performance impact as error correction happens very infrequently, unlike error detection, which generally occurs on every memory access.

Another reason for picking CRC32C is that modern Intel and ARMv8 CPU architectures support calculating CRC32C via instruction intrinsics, therefore providing hardware accelerated performance. When hardware support is not available, a Slicing-by-16 algorithm is used, which has shown a good performance compared to the naive long division method [22].

V. Sparse Matrix Solvers

In this paper we focus our research on ABFT techniques for Sparse Matrix Solvers, however these techniques could be applied to other applications that use similar data structures and data access patterns.

In this section we provide an overview of the TeaLeaf heat diffusion mini-app and describe application specific features that have helped us with creating efficient ABFT techniques.

[1]A burst error is an error affecting contiguous sequence of multiple bits.

A. TeaLeaf

In our research we utilise TeaLeaf, which is a part of Sandia National Laboratories' Mantevo (https://mantevo.org) mini-app benchmark suite. TeaLeaf is a memory bandwidth bound application, meaning that extra memory bandwidth used by ECC has a negative impact on performance. TeaLeaf solves the linear heat conduction equation in 2D on a spatially decomposed regular grid using a five-point stencil. In this paper we focus on using the CG method to perform each time-step, however our ABFT techniques could be used with other solver methods.

Computational steps in TeaLeaf are broken down into *kernels*. We note that the vast majority of TeaLeaf's runtime ($+98\%$) is spent inside three of these kernels, performing matrix-vector products and dot products, and so when designing our ABFT methods these memory access patterns were taken into consideration.

During each time-step when a full CG solve is performed, the sparse matrix does not change, and so we explore how we can leverage this application-specific knowledge in order to reduce the overheads of the ABFT techniques.

B. Sparse Matrices and Dense Vectors

Sparse matrices tend to have a very low number of non-zero elements, and so storing them in compressed formats is much more efficient than storing the whole matrix and performing many redundant calculations on the zero elements. We focus our efforts on the CSR format, where a $m \times n$ sparse matrix is represented by three dense vectors. The first vector v of length NNZ (Number of Non-Zeros) stores all the 64-bit double floating point non-zero elements in row-major order. The second vector y, also of length NNZ, stores the 32-bit column index for each of the non-zero elements. The third vector x of length $m + 1$ stores the 32-bit index into v of the first nonzero element for each row in the matrix.

In [13] it was identified that as long as the matrix dimensions were smaller than $2^{32} - 1$, then elements in the x and y vectors will have unused bits. In previous work, where the dimensions were further restricted to at most $2^{32} - 1$, these unused bits of the y vector were re-purposed to store the redundant ECC data required to protect the v and y vector; however, the x vector was left unprotected. We extend this research by investigating how to additionally protect the x vector with no storage overhead. Note that in many production solvers, the matrix dimensions may be larger than $2^{32} - 1$, warranting the need for 64-bit integer indices; our 32-bit integer techniques are easily extended for this scenario.

Another important data structure that is present in this solver is the double precision floating point vector. Unlike the v vector from the CSR matrix, these double precision floating point vectors have no unused bits that could be used to store redundant data. In our research we therefore investigate how we can combine the redundant data required to provide protection in a manner that does not require extra memory storage.

VI. Protecting Sparse Matrix Solvers

In this section we provide an overview of our new techniques for providing efficient ECC protection to the whole CSR matrix, including the floating point vectors.

A. Protecting CSR Matrices

The previous research in [13] and [20] has already provided the details of how SED, SECDED and CRC32C can leverage application specific knowledge about sparse matrix solvers, such that the redundant bits required by ECC are mixed with the original data to form the necessary codewords to protect the CSR elements from bit flips. We therefore only provide a brief overview of these techniques to demonstrate how the redundant data is embedded into the CSR elements.

A CSR element is formed by pairing two vector elements at the same indices from the double precision floating point v data vector and the 32-bit integer y index vector, forming a 96-bit data structure. As Figure 1 shows, the redundant data required to provide protection for each CSR element is stored in the unused bits of the integer index. This approach puts limits on the matrix size, as when using SED this means that the matrix can have at most $2^{31}-1$ columns, as bigger index values cannot be represented (Figure 1 (a)). Similarly when using SECDED or CRC32C this means that the matrix can have at most $2^{24}-1$ columns (Figures 1 (b) & (c) respectively).

When using CRC32C as the protection method, each row is protected by a single CRC checksum and since only 8 bits can be used from each CSR element, each row has to have at least four non-zero elements; this is not a problem as TeaLeaf has five non-zero elements per row due to the five-point stencil.

1) Protecting the Row Integer Vector: A similar approach for protecting the CSR elements can be applied to protecting the x vector. We note that each element of this 32-bit integer vector can have a value of at most NNZ, and so we can re-purpose the most significant bits again by putting constraints on the size of the matrix.

When using SED, as the Figure 2 (a) shows, the top bit from each element in the index vector is used to store the parity, meaning that the matrix can have at most $2^{31}-1$ elements.

In order to use other ECC techniques, more bits from the integer index vector have to be used. By using the top 4 bits to store the ECC data we can still have $2^{28}-1$ or ≈ 268 million elements in the matrix; however, using any more bits could put too many constraints on the matrix size. Our other ECC techniques require more than 4 bits to store the redundancy bits, and therefore these techniques have to move protect multiple elements at the same time to amortize the ECC bits over more elements. Our new scheme allows us to split the redundancy bits between 2, 4 and 8 elements for SECDED64, SECDED128 or CRC32C respectively. An example of SECDED64 is shown in Figure 2 (b).

2) Less Frequent Correctness Checking: During each time-step of the solve, the sparse matrix does not change between the CG iterations. This means that if an error occurs during an iteration, it will still be there during successive iterations unless another error occurs in the exactly same location. By performing the integrity checks every N accesses to the matrix instead of on every access, we can reduce the cost of performing these checks with a trade off of having to do up to N more iterations of CG before the error is detected. When not performing the ECC checks we still need to make sure that the indices from the CSR matrix are within the correct range so that any out of bounds memory access which could lead to a segmentation fault are avoided. To protect against these, during the iterations where the integrity check is not performed a boundary check is done instead. For the values from the x vector we need to make sure that the values are less than the total number of non zero elements in the matrix, and when accessing the y vector, we need to make sure that the values are less than the number of columns in the matrix. At the end of each time step we also need to perform an extra integrity check for the whole matrix just in case N does not divide the number of iterations performed, to make sure no errors escape unnoticed.

One drawback of this approach is that we lose the ability to correct any errors, since any correctable errors that have been detected during the integrity checks might have been present during the past $N-1$ iterations. This suggests that this approach should only be used with Error Detecting Codes.

B. Protecting Dense Floating Point Vectors

The double precision floating point values do not have any unused bits due to their format, and so in order to provide ECC with no storage overheads we choose to store the redundant data in the least significant bits of the mantissa.

When using SED, the least significant bit in the 64-bit floating point value's mantissa is used to store the parity bit, as shown in Figure 3 (a). More bits need to be used for other ECC techniques: when using SECDED64, SECDED128 and CRC32C, and splitting the redundant bits across 1, 2 and 4 double precision values respectively (Figure 3), then the least significant 8, 5 and 8 bits respectively are used to store these redundant bits. The redundant bits from SECDED128 or CRC32C have to be split across multiple vector elements in order to reduce the amount of noise that this storage method would otherwise introduce into the data. The storage of the redundant data in the least significant bits poses a risk that the solver may take longer to converge, or in the worst case, the solver might fail to converge altogether. Work by Elliott et al. in [23] showed that solvers can successfully converge without requiring additional iterations in an event of a single bit flip, which implies that error detection using SED has no effect on the accuracy of the solver. To control the level of noise and bias caused by the storage of redundancy bits in the floating point values, our framework masks all these bits to 0 whenever a floating point value is used for computation. In our experiments using SECDED64, SECDED128 and CRC32C, the solver has always converged with the norm of the solution vector within $2.0 \times 10^{-11}\%$ of the expected answer. However, on some problem sizes the number of iterations increases in the later time-steps, but the increase in the total number of iterations was always observed to be less than 1%.

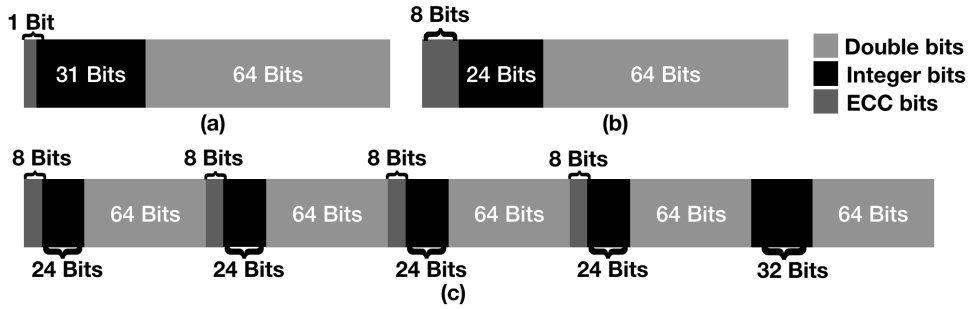

Fig. 1. Storing redundant data in unused index bits in sparse matrix elements. Sub-figures **(a)** and **(b)** show how SED and Hamming Code (respectively) are used to protect each CSR element. Sub-figure **(c)** demonstrates how our CRC32C checksum is distributed across the whole matrix row.

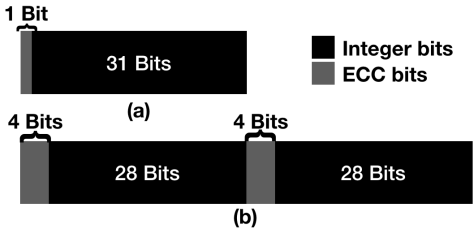

Fig. 2. Storing redundant data in unused index bits in the row vector from the sparse matrix. Sub-figure **(a)** shows how Parity is used to protect an individual vector element. Sub-figure **(b)** shows how the redundant data is distributed across multiple vector elements.

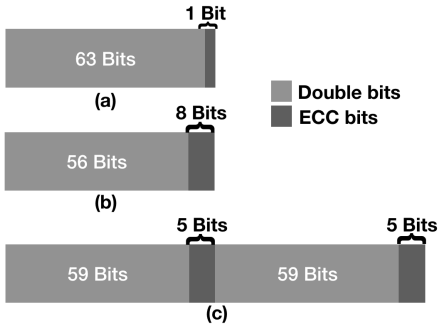

Fig. 3. Storing redundant data in the least significant bits of the mantissa. Sub-figures **(a)** and **(b)** show how Parity and Hamming Codes (respectively) are used to protect an individual vector element. Sub-figure **(c)** shows how the redundant data is distributed across multiple vector elements.

C. Read Modify Writes and Caching

Using multiple vector elements to form a codeword raises problems with access granularity and performance. When modifying a value in a vector, a potential Read-Modify-Write (RMW) has to be performed as only part of the codeword is being modified, and so the unmodified vector elements need to be read and used to recalculate redundancy bits. This operation is quite expensive as the read needs to perform the integrity check on the data currently stored and the write needs to recalculate the redundancy data, resulting in two ECC calculations every write. We can remove most of the RMWs by noticing that when performing calculations at position i, the Sparse Matrix Solve algorithm will then work on the next element $i + 1$, meaning that if we buffer the writes we can commit a whole ECC element to memory in one go, performing a single integrity calculation per multiple writes. This approach however requires the algorithm to be adapted so that the calculations performed are not performed on individual vector elements, but instead on the whole ECC element at a time. This approach dramatically reduces the overhead as the integrity calculation on the read is no longer needed. It also avoids race conditions in parallel implementations as threads never write to the same ECC element.

Another issue with these compound ECC elements is that usually when an element at index i is being read, the element at index $i + 1$ is likely to be accessed at the next iteration of the algorithm. In order to avoid having to recompute the same integrity check multiple times we buffer each ECC element so that the neighbouring vector values are readily available.

This buffering of reads removes most of the unnecessary duplicate integrity checks, except for the accesses in the sparse matrix-vector product kernel. This is because TeaLeaf uses a five-point stencil, meaning that when calculating the product at index (i, j), the vector elements at the following indices are accessed:

- $i + (j - 1) * n$
- $i - 1 + j * n$
- $i + j * n$
- $i + 1 + j * n$
- $i + (j + 1) * n$

This access pattern means that at least 3 ECC compound elements are accessed, and so the buffering scheme for ECC elements provides no real benefits. To combat this we have created a caching scheme within the sparse matrix-vector product kernel that is both multiple ECC element and multi-iteration aware.

These buffering techniques come with trade offs as although the performance is improved, our techniques described here protected the program from bit flips in both main memory and cache memory. These small caching buffers however might be subject to bit flips in the cache memory, and since accessing these buffers does not perform integrity checks, then only the data stored in the main memory is fully protected.

VII. Performance Results

Most of the ABFT techniques we have described provide the same level of protection as the alternatives commonly found

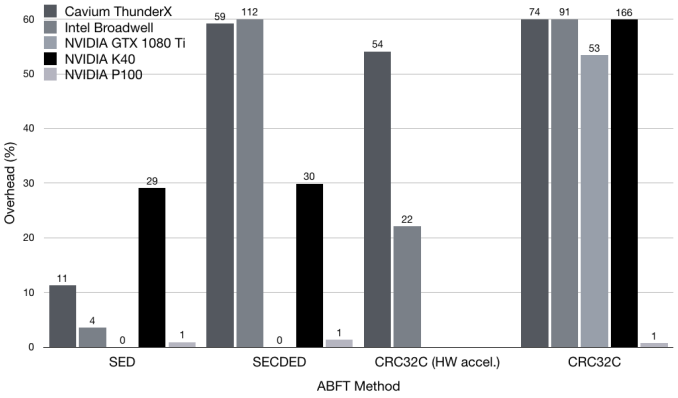

Fig. 4. Execution time overheads for the ABFT techniques for protecting CSR elements.

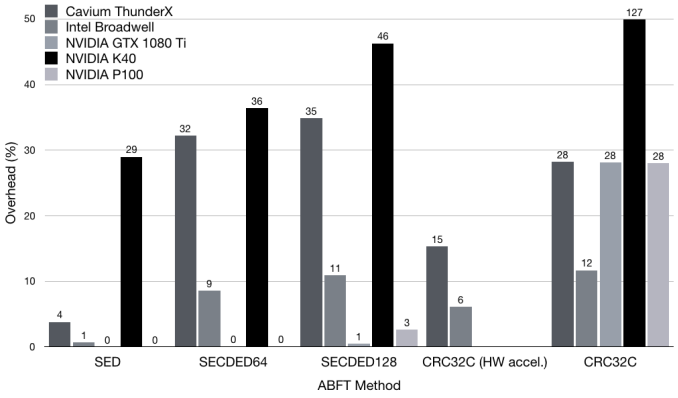

Fig. 5. Execution time overheads for the ABFT techniques for protecting the row integer vector used by the CSR format.

in hardware. The exception is our CRC32C technique, which if used to provide detection rather than correction, can detect many more bits flips than the ECC hardware that is common today. We have ported our techniques to different architectures to measure their relative runtime overheads, and so in this section we discuss our findings and identify strengths and weaknesses of these different approaches in order to choose the most advantageous protection for a given platform.

We performed our experiments on the following platforms:

- The results for Intel Broadwell were obtained using a dual socket node with 18 Core Intel Xeon CPU E5-2695 v4 CPUs and the Intel Compiler v2017 with OpenMP;
- The results for Cavium ThunderX were obtained using a dual socket node with 48 Core Cavium ThunderX CPUs and the ARM HPC Compiler 1.3 with OpenMP;
- The results for the NVIDIA GPUs (K40, GTX 1080 Ti, P100) were obtained with CUDA 8 and gcc v4.9.4.

We used an input file for TeaLeaf with 2,048x2,048 cells, performing 5 timesteps. All tests were run five times with the mean time taken.

The Intel Broadwell and NVIDIA P100 platforms provide hardware ECC which cannot be turned off. The platforms for the Cavium ThunderX and NVIDIA GTX 1080 Ti consumer GPU provide no hardware ECC. The NVIDIA K40 HPC GPU allows the system administrator to turn the hardware ECC on and off, and so this platform was used to set the target runtime overhead. The hardware ECC on this GPU incurs a measured overhead of 8.1% for TeaLeaf, due to the fact that TeaLeaf is a memory bandwidth bound application and this ECC method requires some of the bandwidth for the redundancy data.

A. CSR Matrix Protection Overheads

The results in Figure 4 present reduced overheads relative to the techniques shown in [13] and [20]. Our latest results agree with our previous findings, showing that SED provides good performance on almost all of the platforms, with the notable exception of the K40.

The poor results on the NVIDIA K40 for all our ABFT techniques are due to low occupancy, as the high register count required by our software ECC techniques has resulted in a low number of warps per Streaming Multiprocessors (SMs). These results have significantly improved with the recent NVIDIA Pascal architecture GPUs (NVIDIA P100 and GTX 1080 Ti), where the higher register count resulted in a better utilisation of the GPU. Our previous research found SECDED to cause very high overheads, which we considered to be impractical. However, on the NVIDIA Pascal GPUs these techniques cause an overhead of less than 1%, which is much more acceptable. Another promising result is the 1% overhead for CRC32C on the NVIDIA P100 GPU, significantly lower than all other platforms we tested. The overheads for SECDED and CRC32C on other platforms are still quite large, and reducing the cost of these in software only might not be possible.

The results in Figure 5 show the overheads for the proposed ABFT techniques to protect the dense integer vector required by the CSR matrix storage format. Similar to the previous results, SED incurs little overhead on most platforms. These results also show there are no benefits of using SECDED128 over SECDED64 for CSR matrix protection as the latter provides better performance results with higher resiliency.

The different methods of protecting the CSR elements and the CSR x vector can be mixed together to fully protect the whole matrix, with the overhead being approximately equal to the sum of the overheads of the two techniques. These findings are promising as we are able to protect the whole CSR matrix with minimal overhead. For example, when protecting the whole matrix with SED or SECDED(64), we add less than 2% runtime overhead on both NVIDIA GTX 1080 Ti and P100. Using hardware accelerated CRC32C we are also able to protect the whole matrix with a 30% runtime overhead on the Intel Broadwell platform, showing how instruction set support can aid the ABFT techniques.

As previously described, the overheads for protecting the CSR matrix can be further reduced by performing the integrity checks less frequently, i.e. not on every iteration. In Figure 6 we can see that at first performing the checks every other iteration proves beneficial for SED on Intel Broadwell, however performing even less frequent checks gives no further

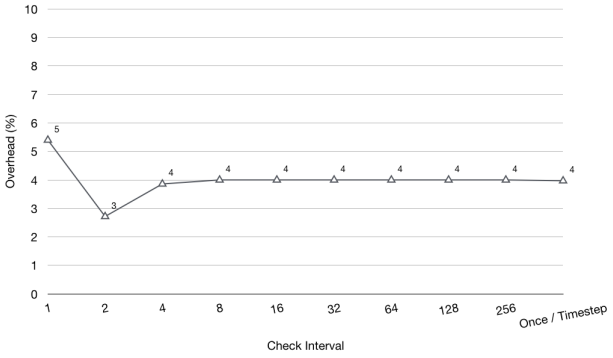

Fig. 6. Runtime overheads of protecting the whole CSR matrix on Intel Broadwell with SED with different check intervals, measured in iterations.

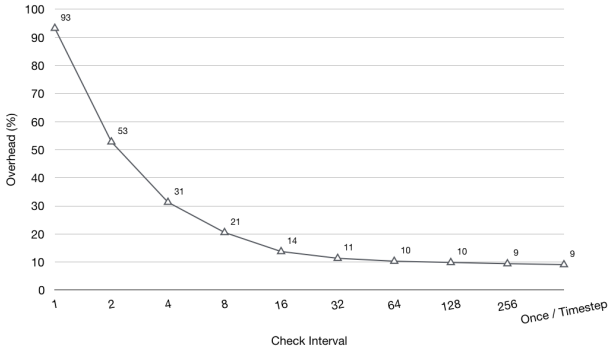

Fig. 7. Runtime overheads of protecting the whole CSR matrix on Cavium ThunderX with Hamming Codes (SECDED64) with different check intervals, measured in iterations.

benefit. This overhead comes from the range checks for the index values in the CSR matrix which prevent segmentation faults, which introduce a fixed cost of some extra branching. We can similarly reduce overheads of other combinations of ABFT techniques on Intel Broadwell, however none of them achieve below a 4% runtime overhead.

Figure 7 shows the overhead of protecting the CSR matrix using SECDED on the Cavium ThunderX platform. The trend shows a similar pattern to the results for Intel Broadwell, where the less frequent checks can be used to reduce the

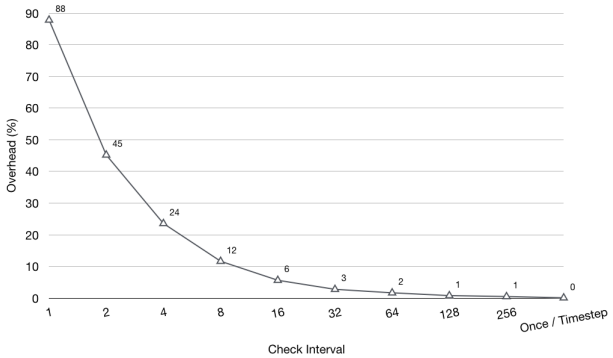

Fig. 8. Runtime overheads of protecting the whole CSR matrix on NVIDIA GTX 1080 Ti with CRC32C with different check intervals, measured in iterations.

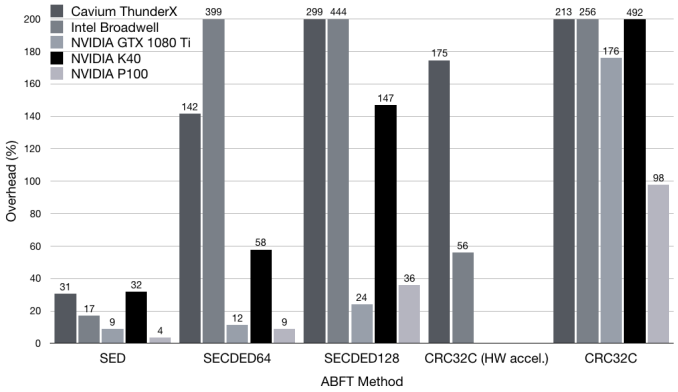

Fig. 9. Runtime overheads for the ABFT techniques for protecting the dense double precision floating point vectors.

overheads down to just 9%, after which most of the overhead comes from performing range checking. Figure 8 shows that by performing these checks only every 128 iterations of the CG solve, we are able to reduce the overhead of protection using CRC32C on the consumer GPU from 88% to just 1%, demonstrating how successful this method can be.

B. Dense Floating Point Vector Protection Overheads

We expected the overheads for protecting the dense double precision vectors to be much greater than the overheads for protecting just the CSR matrix. This was because the matrix is not modified during the time step and it is only accessed by one kernel to calculate the sparse matrix-vector product.

The results in Figure 9 show the runtime overheads of our techniques for fully protecting all of the double precision floating point vectors in TeaLeaf. By choosing to protect the vectors using SED, we incur an overhead of between 4% and 32%, depending on the platform, showing that minimal mechanism for protection against SDC can be efficiently implemented in software. The NVIDIA GPUs have also shown very good performance for SECDED64, with overheads of just 12% and 9% for the GTX 1080 Ti and P100 respectively.

By combining these results with the previous findings for protecting the CSR matrix, we have been able to demonstrate a software based ECC scheme which fully protects the matrix and the double precision floating point vectors using SECDED with an overhead of approximately 11%, getting close to our 8.1% target. At the same time, we can choose protection schemes which can detect a greater number of bit flips than current hardware, an ability which might be increasingly important as we grow towards Exascale supercomputers.

VIII. Discussion and Conclusions

In this paper we have demonstrated efficient ABFT techniques to fully protect the data stored by a sparse matrix solver from bit-flips by leveraging our knowledge about the application, something that hardware ECC alone cannot do.

Our software-based ECC approach has proved successful, as we have been able to protect all the data with a runtime overhead as low as 11% on some of the platforms, while

providing a similar level of resiliency to current hardware. Our techniques demonstrate a clear advantage over current hardware ECC and Chipkill approaches, as ours require no extra storage or DRAM devices. We are also able to detect a much larger number of bit flips than current hardware. Our proposed solutions have been able to protect the data stored by the sparse matrix solver on devices which do not provide hardware ECC, such as consumer GPUs. Being able to turn off ECC in hardware could therefore be an advantage in future Exascale systems, especially if the dedicated memory and bandwidth could then be given to the application instead. These ABFT methods also allow the program to decide what should happen in the event of an uncorrectable error, as the CG solve might be able to continue due to its iterative nature, without the need for checkpoint-restart, whereas the hardware alternative would always issue an error.

However, we have not yet been able to achieve low runtime overheads on all of the platforms which were tested. These techniques also require modifications to the code, which ideally would be implemented at the library level, in packages such as PETSc (https://www.mcs.anl.gov/petsc/) or Trilinos (https://trilinos.org/), which we hope to explore in future work.

We have shown that hardware accelerated CRC32C calculations were an improvement over software-only solutions, demonstrating that instruction set design can help with achieving better performance, and that combining software and hardware methods to protect against SDCs might prove beneficial.

To conclude, we have shown that sparse matrix solvers can be efficiently protected using ABFT techniques that utilise software ECC at no extra cost in terms of memory storage, saving memory bandwidth which was previously required to move the explicitly stored redundant ECC data. We have shown that understanding the data access patterns, such as the five-point stencils in TeaLeaf, can be beneficial to the performance of the resilience techniques. We have also shown that software techniques can provide a higher degree of protection from bit flips than current hardware approaches, and if some trade-offs on how quickly errors are detected can be made, these techniques can be applied with runtime overheads as low as 4% on CPUs and 1% on GPUs.

ACKNOWLEDGMENTS

The authors would like to thank EPSRC for funding this research. We also extend thanks to the Intel Parallel Computing Centre at the University of Bristol, for providing access to the Zoo testbed, and to GW4 for providing access to their Tier 2 Isambard supercomputer. This work was also supported by funding from the European Communitys Seventh Framework Programme [FP7/2007-2013] under the Mont-blanc 2 Project, grant agreement n^o 610402.

REFERENCES

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems, Peter Kogge, editor & study lead," 2008.

[2] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of DRAM raw error rate on a supercomputer," *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, vol. 00, pp. 645–655, 2016.

[3] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," *Communications of the ACM*, vol. 54, no. 2, p. 100, 2011.

[4] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[5] V. Sridharan, J. Stearley, N. Debardeleben, S. Blanchard, and S. Gurumurthi, "Feng Shui of supercomputer memory positional effects in DRAM and SRAM faults," *SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2013.

[6] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems The Good , The Bad , and The Ugly," *ASPLOS '15 Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 297–310, 2015.

[7] L. B. Gomez, F. Cappello, L. Carro, N. DeBardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda, "GPGPUs: How to combine high computational power with high reliability," *Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–9, 2014.

[8] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. Debardeleben, P. Navaux, L. Carro, and A. Bland, "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation," *2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA 2015*, pp. 331–342, 2015.

[9] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on GPUs in the field," *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2016-April, pp. 519–530, 2016.

[10] J. F. Ziegler and W. A. Lanford, "Effect of cosmic rays on computer memories," *Science*, vol. 206, no. 4420, pp. 776–788, 1979.

[11] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[12] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Microelectronics Division*, pp. 1–23, 1997.

[13] S. McIntoshSmith, R. Hunt, J. Price, and A. W. Vesztrocy, "Application-based fault tolerance techniques for sparse matrix solvers," *The International Journal of High Performance Computing Applications*.

[14] "Challenges in thermal management of memory modules," https://www.electronics-cooling.com/2008/02/challenges-in-thermal-management-of-memory-modules/, accessed: 2017-07-05.

[15] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," *SIGPLAN Not.*, vol. 45, no. 3, pp. 397–408, Mar. 2010.

[16] L. Borucki, G. Schindlbeck, and C. Slayman, "Comparison of accelerated DRAM soft error rates measured at component and system level," in *2008 IEEE International Reliability Physics Symposium*, April 2008, pp. 482–487.

[17] "TOP500 - Piz Daint," https://www.top500.org/system/177824, accessed: 2017-07-25.

[18] D. McMorrow and M. Corporation, *Technical Challenges of Exascale Computing*. MITRE Corporation, 2013.

[19] S. McIntosh-Smith, M. Martineau, M. Boulton, W. Gaudin, P. Garrett, W. Liu, and R. Smedley-Stevenson, "The TeaLeaf heat diffusion benchmark from Mantevo," https://github.com/UoB-HPC/TeaLeaf, accessed: 2017-06-30.

[20] J. Yeh, G. Pawelczak, J. Sewart, J. Price, A. Avila Ibarra, S. McIntosh-Smith, L. Bautista-Gomez, and F. Zyulkyarov, "Software-level Fault Tolerant Framework for Task-based Applications," *Poster session presented at IEEE/ACM SuperComputing, Salt Lake City, United States*, 2016.

[21] P. Koopman, "32-Bit Cyclic Redundancy Codes for Internet Applications," 2002.

[22] "Fast CRC32," http://create.stephan-brumme.com/crc32/, accessed: 2017-06-30.

[23] J. Elliott, F. Mueller, M. Stoyanov, and C. Webster, "Quantifying the impact of single bit flips on floating point arithmetic," *preprint*, 2013.