View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Explore Bristol Research

McIntosh–Smith, S., Hunt, R., Price, J., & Vesztrocy, A. W. (2018). Application-based fault tolerance techniques for sparse matrix solvers. *International Journal of High Performance Computing Applications*, *32*(5), 627-640. https://doi.org/10.1177/1094342017694946

Peer reviewed version

Link to published version (if available):
10.1177/1094342017694946

Link to publication record in Explore Bristol Research
PDF-document

This is the author accepted manuscript (AAM). The final published version (version of record) is available online via Sage Publications at https://journals.sagepub.com/doi/10.1177/1094342017694946. Please refer to any applicable terms of use of the publisher.

University of Bristol - Explore Bristol Research

General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available: http://www.bristol.ac.uk/pure/about/ebr-terms

Journal Title
XX(X):1–11
©The Author(s) 2016
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/

Application-Based Fault Tolerance Techniques for Sparse Matrix Solvers

Simon McIntosh-Smith[1], Rob Hunt[1], James Price[1] and Alex Vesztrocy[2]

Abstract

High-performance computing (HPC) systems continue to increase in size in the quest for ever higher performance. The resulting increased electronic component count, coupled with the decrease in feature sizes of the silicon manufacturing processes used to build these components, may result in future Exascale systems being more susceptible to soft errors caused by cosmic radiation than current HPC systems. Through the use of techniques such as hardware-based error-correcting codes (ECC) and checkpoint-restart, many of these faults can be mitigated, but at the cost of increased hardware overhead, run-time, and energy consumption that can be as much as 10–20%. Some predictions expect these overheads to continue to grow over time. For extreme scale systems, these overheads will represent megawatts of power consumption and millions of dollars of additional hardware cost, which could potentially be avoided with more sophisticated fault-tolerance techniques. In this paper we present new software-based fault tolerance techniques that can be applied to one of the most important classes of software in HPC: iterative sparse matrix solvers. Our new techniques enables us to exploit knowledge of the structure of sparse matrices in such a way as to improve the performance, energy efficiency and fault tolerance of the overall solution.

Keywords

Fault tolerance; sparse matrix algorithms; Exascale; single event upsets; error-correcting; bit-flips

Introduction

As the microprocessors and memory devices from which supercomputers are assembled adopt silicon manufacturing processes with ever smaller feature sizes, the likelihood of errors occurring in data sets due to transient faults increases. There are a number of reasons for these faults: they could be due to physical faults in the underlying hardware, faults in the software, or even transient errors due to a silicon device being struck by cosmic radiation or other electromagnetic interference. Memory devices are one of the main victims of this last category of transient error. According to Zeigler and Lanford (1), errors affecting memory devices can be divided into two basic groups: hard errors are those caused by a physical defect, while soft errors are transient in nature and caused by some kind of electromagnetic interaction, such as a cosmic ray strike. Considerable work has been carried out on understanding the causes and effects of cosmic rays on silicon devices (1; 2; 3; 4), in particular on their effect on DRAM devices (5; 6; 7; 8).

A 2009 study by Schroeder et al at Google recorded between 2,000 and 6,000 memory errors per GByte of DRAM per year (9). DRAM cells are now largely resistant to faults, due to the design of their capacitor-based cells (10). While these errors are increasingly rare per DRAM device, the sheer number of these devices in an Exascale system means that this class of soft error will always be a concern. Beyond DRAM, contemporary processors now include tens of megabytes of on-chip SRAM, which, as a transistor-based memory technology, is increasingly prone to errors caused by cosmic rays as the size of transistors continues to shrink. In practice, the use of error correcting code (ECC) hardware improves the reliability of computer systems by detecting and correcting single bit errors (historically accounting for 98% of all memory errors (11)), as well as detecting (but not correcting) double bit errors.

Despite the refining of ECC hardware mechanisms over the years, the hardware and additional storage required for ECC carries a silicon area and energy cost overhead that is non-trivial. For example, a typical single error correct, double error detect (SECDED) hardware implementation for a memory will require 8 bits of additional storage for each 64 bits of data, representing a 12.5% storage overhead for supporting ECC. In addition, the memory controller adds complexity in order to calculate the appropriate 8 parity bits whenever a 64-bit location is written to, and has to check the 8 parity bits whenever a 64-bit location is read from.

The additional hardware complexity and bandwidth requirement increases the energy required for every memory access by at least 12.5%. Some implementations may provide the ECC support within the memory controllers themselves, but even in this case, there is 12.5% more data to move around, and this will take at least 12.5% more energy to move these extra bits. At Exascale, where the first systems have the challenging goal of using no more than

[1]University of Bristol, UK
[2]University College London, UK

Corresponding author:
Simon McIntosh-Smith, Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, U.K.
Email: simonm@cs.bris.ac.uk

20MW of power, it is estimated that memories will consume 20% of this power, while the processors will be responsible for 52% of the total power consumption (12). Of this 52% used by the processors, one fifth is expected to be consumed by on-chip memories and off-chip memory bandwidth, and so in total, memory access will account for roughly 30% of the 20MW power budget of the first Exascale systems. Therefore, being able to reliably compute without the need for ECC hardware could present a distinct advantage in terms of an Exascale system's energy efficiency. Our approach also benefits application performance, as errors caused by bit-flips will be identified — and in many cases corrected — within the iterative sparse matrix solver itself, saving the substantial performance penalty that would have otherwise been necessary for an operating-system level ECC recovery scheme or for a machine-wide checkpoint-restart.

In this paper we focus on sparse matrix calculations, where explicit matrix cell location data must be stored along with the respective matrix cell values. This is in contrast to dense matrix computations of dimension $m \times n$, where the address of any element $d_{i,j}$ in the array can be calculated simply from the base address, i, j, m and n. Thus dense matrices are stored as contiguous regions of memory, with the addresses of required cells trivially computed as they are required. With sparse matrices, typically only a small fraction of the matrix elements are non-zero, and so the matrix is stored as corresponding arrays of indices and values. Therefore, in a two-dimensional sparse matrix with one double-precision floating-point value and two 32-bit unsigned integer indices per non-zero element, there can be as much data to describe the structure of the matrix (two 32-bit indices per non-zero) as there is for the actual values of the non-zero elements of the matrix. Furthermore, when binary (or *pattern*) matrices are considered, the indices themselves represent *all* of the stored data.

Whilst there are iterative algorithms for sparse matrix calculations that are known to be tolerant of some errors caused by bit-flips in the element *values* (13), there is almost no existing literature which considers the effect of bit-flips on the element indices, which as we have established, can themselves represent up to half of the data in a sparse matrix – for applications which use sparse matrix solvers, the sparse matrices might typically be the majority of all data in memory, and thus any soft error that does affect a running sparse matrix solver is likely to affect the sparse matrix itself, with roughly equal probabilities of a soft error effecting the index data vs. the value data. An uncorrected bit-flip in an index within a sparse matrix structure could cause a catastrophic failure, potentially resulting in a memory fault and a subsequent checkpoint-restart recovery sequence. Perhaps worse, this kind of data corruption could instead result in a silent error, where incorrect results might erroneously be believed to be valid. This class of sparse-matrix memory index error has largely been overlooked in the literature to date, yet for any bit-flip affecting a sparse matrix, that bit flip is equally as likely to occur in the 64-bits of index data as it is in the 64-bits of value data.

Contributions

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

1. We present new application-based fault tolerance (ABFT) techniques for exploiting spatial relationships in sparse matrices to help ensure the correctness of indexing data in the presence of bit-flips.
2. We present the results of a statistical analysis of the efficacy of our new fault tolerance techniques when applied to the set of sparse matrices held in the University of Florida's sparse matrix collection (14).
3. We describe a software-based error correcting code scheme that protects the entire sparse matrix, while requiring less space and overhead than the equivalent hardware ECC scheme.
4. We detail performance results for a modified, CG-based sparse iterative solver, which describe the relative overheads of each of the software-based fault tolerance techniques we have developed.

The rest of this paper is structured as follows. In "Previous Work", we provide a brief overview of the latest research relevant to our work. In Section "Sparse Matrces" we then begin our focus on sparse matrices, the backdrop and impetus to our contribution. We look at their structure and storage formats, and how we can exploit both of these in turn. In Section "Sparse Matrix Index Constraints" we develop a range of constraints that enable the detection and correction of a range of bit-flip errors in sparse matrix indices. The first results section, "Efficacy of the Sparse Matrix Constraints Scheme", evaluates how well these techniques actually protect against bit flips. "Software-based Error Correcting Codes" details our second set of fault tolerance techniques, which use Hamming techniques to protect both the indices and data in the matrices. "Performance Overheads for Software Fault Tolerance Schemes" measures the relative costs of these techniques, before we concludes and discusses ideas for future work in "Discussion and Conclusions".

Previous Work

While little literature has yet addressed the issue of errors in sparse matrix data, there is a growing body of work exploring the potential benefits of software-based fault tolerance techniques. Recent work by Hukerikar et al on software-based resilience to bit-flips has explored high-level, transparent techniques that enable a software developer to specify which variables and operations within their code *must* be performed in a resilient manner, and which have some degree of natural fault tolerance (15). Mitigating the effects of bit-flips through the use of type qualifiers and a library of resilient functions, this approach allows the user to tailor their code to either ensure accuracy, or to mask those bit-flips which would have a negligible effect on their program. In Hukerikar et al's work, the focus is on masking bits where a flip can be tolerated — in the unused most significant bits of the index data or in the relatively insignificant lowest bits of the floating-point data. By contrast, in our work we use the constraints that naturally arise due to the structure inherent in most sparse matrices to identify when a bit flip has occurred.

While no previous work has directly considered the effect of bit-flips on sparse matrix index data, Elliott and Mueller have analyzed the effects of bit-flips on floating-point values (16). Their work showed that a bit-flip in a floating-point value has a high probability of occurring in the least significant bits of the fraction, and thus many of these events would likely result in small rounding errors which have little effect in many (but not all) kinds of scientific calculation. In our work we apply a similar approach but to the index values of the matrix, and then develop mechanisms to spot and potentially correct such errors when they occur.

Other work by Maruyama et al has looked at the need for fault tolerance in software running on commodity GPUs (17), which are increasingly being used to help accelerate HPC applications. Commodity GPUs, unlike their higher-end HPC counterparts, tend not to include hardware ECC support, and so any bit-flips that occur in the GPU DRAM or on-chip SRAM are potentially a serious problem. To address this issue, Maruyama et al have developed schemes that combine elements of software ECC and grid-based parity check bits with checkpoint-based methods. Our work differs in that it presents solutions for sparse matrix methods running on any hardware platform. Our methods can be used to help augment alternatives to ECC hardware solutions, efficiently protecting sparse matrix indices and values, with no additional storage overhead and low performance overhead. As such, our methods can benefit any processors without hardware ECC support, such as commodity GPUs or consumer-level CPUs.

Sparse Matrices

Throughout rest of this paper, we work with Fortran-style numbering of arrays, i.e., an array of length N starts with element 1 and runs to element N.

The amount of data used to store the row and column locations in sparse matrices is significant yet often overlooked. From a study of over 2,600 sparse matrices from the University of Florida's Sparse Matrix Collection (14), the average percentage fill of a given sparse matrix was $\sim$2%, with the median fill an order of magnitude lower at $\sim$0.24%. This low fill rate is why it is much more efficient to store only the non-zero elements in a sparse matrix, and thus why they are commonly stored in compressed formats, such as the so-called Coordinate (COO) format or the Compressed Sparse Row (CSR) format, which both store location data for each (non-zero) matrix element, along with the (non-zero) matrix values themselves.

In the rest of this section we present a detailed analysis of the effects of bit-flips on sparse matrix index data, before moving on to develop new techniques for detecting and correcting these errors which require no additional storage and have a low performance penalty. We believe that this is the first time that errors in sparse matrix index data have been addressed in the literature.

Sparse Matrix Storage Formats

The simplest sparse matrix storage scheme is the COO (Coordinate) format (18), whilst the CSR (Compressed Sparse Row) format is often used for its advantages over COO in terms of space and simplicity in indexing.

$$A = \begin{pmatrix} 2.4 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3.5 & 0.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 1.8 & 0.0 \\ 0.0 & 0.0 & 0.0 & 3.3 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.7 \end{pmatrix}$$

Figure 1. An example sparse matrix.

In COO, when working with two-dimensional matrices, there are three arrays of length NNZ (Number of Non-Zeros): two 32-bit unsigned integer arrays to hold the index dimensions and a (single- or double-precision) floating-point array to hold the value at each non-zero point. (If the matrix is only filled with integers, this third array could also be an integer array, whilst if it is a binary or pattern matrix, there is no need for the third array at all.)

In CSR, when working with two-dimensional matrices, there are two arrays of length NNZ whilst the third is of length $m + 1$ (for a matrix of dimension $m \times n$). As before, we have one array that stores the values at each non-zero point, whilst the other two arrays store sufficient data to calculate the index dimensions. In this case, the index array of length NNZ holds the column index of each non-zero element. The $(m + 1)$-length array stores the position in the other two arrays of each element that starts a new row, with the $(m + 1)$ element always set to $\mathrm{NNZ} + 1$.

We can illustrate the COO and CSR schemes by considering the matrix in Fig. 1. In the COO scheme, we represent the non-zero elements using two coordinate vectors, x and y, as well as the array of non-zero values, v, to which x and y directly correspond (i.e., $v[i]$ is located at $(x[i], y[i])$). Here we store the matrix A as in Table 1.

v	2.4	3.5	4.0	1.0	1.8	3.3	1.0	0.7
x	1	2	2	3	3	4	4	5
y	1	2	4	3	4	4	5	5

Table 1. The sparse matrix A in the COO format.

In the CSR scheme, we again have an array of values, v, as well as two coordinate vectors, but one of the coordinate vectors (x_{off}) represents the m offsets into v of the starting elements of each row, along with the $(m + 1)$ element storing the value $\mathrm{NNZ} + 1$. This gives us $v[i]$ located at $(j, y[i])$, where j is the unique integer that satisfies the relation $x[j] \leq i < x[j + 1]$. Thus we store the matrix A in the CSR scheme as in Table 2. The CSR scheme assumes that there are no empty rows in the matrix.

v	2.4	3.5	4.0	1.0	1.8	3.3	1.0	0.7
x_{off}	1	2	4	6	8	9		
y	1	2	4	3	4	4	5	5

Table 2. The sparse matrix A in the CSR format.

In the next section we will show a range of valid index constraints that naturally arise from the structure often inherent in sparse data sets. We will initially present our new methods in relation to the simpler COO format, before showing how they extend to CSR.

Sparse Matrix Index Constraints

Exploiting Sparse Matrix Size Constraints

A first constraint that must always apply is that the indices i, j corresponding to each element a_{ij} of an $m \times n$ sparse matrix A must have a value between one and the size of their dimension. Intuitively, this is obvious: assuming that each non-zero element e_{ij} is located at (x_k, y_k), where $x_k = x[k] = i$ and $y_k = y[k] = j$, then

$$0 < x_k \leq m \tag{1}$$

$$0 < y_k \leq n. \tag{2}$$

While this might not at first appear to be that useful, an analysis of the University of Florida's collection of ∼2,600 sparse matrices shows that the largest dimension of any of their sparse matrices is ∼118 million, which requires unsigned indices of 27 bits. However, matrices of this size are rare in the collection, and excluding the largest 13 matrices reduces the maximum dimension to 16,777,216 (i.e., 2^{24}), which needs only 24 bits for representation. The important implication is that while there are 2^{32} valid unsigned integers, the range of valid indices is much smaller, the exact number depending on the maximum size of the matrices under consideration.

We can apply this constraint to immediately improve the fault tolerance of a sparse matrix-based computation. As a sparse matrix is processed, we check that each of the indices still satisfies their relevant constraint as defined in either Equation 1 or 2. This method will automatically detect any bit-flips that have occurred in the most significant part of each index: those in the bit positions higher than those used to represent the size of each dimension. In our examples from the Florida collection, this approach will detect bit-flips in at least 5 of the 32 index bits (15.5% of potential index bit-flips), or at least 8 of the index bits (25.0%) if we exclude the largest 13 matrices. The method is even more effective for smaller matrices.

In addition, these single-bit flips, once detected by the constraint, could also be corrected in a similar manner to the masking used by Hukerikar et al; zeroing the desired number of bits in the most significant 'guarded' range of the index would restore the correct index value, and the sparse matrix calculation could continue without interruption, and without resorting to a checkpoint-restart sequence. When the size of each dimension is not an exact power of 2, these constraints will detect more errors than simply masking the top bits; whilst the index cannot be guaranteed to be corrected, the error can be detected if it violates one or both of the constraints.

The overhead of checking the constraints is low. The constraint checking can be implemented as either a simple conditional test requiring a 32-bit unsigned integer comparison, or it can be implemented as a bitwise test, whichever is fastest on the target architecture. This constraint checking can also be performed in parallel with other parts of the sparse matrix computation. For example, if one were to assume that bit-flips in the indices would not result in memory faults, but would merely index the wrong data, then one could envisage an implementation where the indices of an element would be used to access the element value, likely requiring a long-latency memory load from DRAM, during which time the indices could be checked against the relevant constraints. If during the constraint checking an error was found, the indices could be corrected and the element value load reissued, with the erroneous element value discarded. In general, it should be possible to overlap constraint checking with element value loads, as the latter are likely to be cache misses and require long latency DRAM accesses. Our new approach should be considerably faster than checkpoint/restart schemes, as only a few simple tests are needed and only a small, local correction required to a single 32-bit index in the event of an error.

Exploiting Sparse Matrix Ordering Constraints

Having established that we can use a sparse matrix's size to define constraints on valid element indices, and that these constraints can subsequently be used to detect and potentially correct bit-flips in the index data, we can now look for more opportunities to establish further constraints that would enable us to detect a wider range of errors.

The first set of constraints was simple and considered only the matrix size. A second simple yet useful set of constraints can be applied to sparse matrices in the COO format when the non-zero elements are stored in a consistent order. Assuming the indices are always in increasing order (an assumption that is true for all ∼2,600 sparse matrices in the University of Florida collection), this gives a monotonically non-decreasing sequence for the major dimension and, within that, a strictly increasing sequence for the minor dimension (the wrap-arounds when the major dimension increases being the only exceptions to this constraint). If matrices are stored in this fashion, it gives us two new simple constraints that must always be true:

$$x_{k-1} \leq x_k \leq x_{k+1} \tag{3}$$

$$y_{k-1} < y_k \quad \text{when} \quad x_{k-1} = x_k \tag{4}$$

where $1 < k < \text{NNZ}$.

These constraints will be useful if a bit-flip in an index causes it to break the apparent ordering of the indices. The probability that a bit-flip in an index will cause a violation of these new constraints depends on the nature of the indices in real sparse matrices. In practice, constraints 3 and 4 can provide significant additional index data error detection and, potentially, an aid to correction. Further, the major index has even greater protection since its non-decreasing sequence changes much more slowly and — for non-degenerate matrices — changes by at most one. This means that the vast majority of bit-flips in the major index (31 out of 32, or ∼97%) would violate a constraint and would therefore be detectable and trivially correctable.

So far we have only considered sparse matrices in the COO format. When we are dealing with matrices in the CSR format, we can add to the above constraints since we would have strict inequality in the x_{off} array:

$$x_{\text{off}}[k] < x_{\text{off}}[k+1] \tag{5}$$

where $1 < k \leq m$, with m the number of rows. However, it should be noted that in this case the changes to x_{off} will be at least one, and often greater.

Ordering constraints 3 and 4 can be implemented in a similar manner to the simple dimensional constraints (1 and 2) described earlier, using simple 32-bit unsigned integer comparisons, executed concurrently with the element value accesses themselves. When an ordering constraint violation is detected, it is potentially possible to correct the error. If we assume that only a single bit has been flipped in the index that has violated the ordering constraint, in many cases it will be possible to determine which bit has caused the violation.

Exploiting Sparse Matrix Symmetry Constraints

So far the constraints we have developed have depended only on the size and ordering of the sparse matrix elements; the constraints have not considered any other structure that might be present in the data. From a statistical analysis of the sparse matrices in the University of Florida's collection, we see that there is often significant structure that underlies the matrix data; that is, the non-zeros in the vast majority of sparse matrices are not simply randomly distributed across the matrix. For a number of numerical algorithms, such as those derived from stencil-based methods, this is immediately obvious, with useful structure such as symmetry in the associated sparse matrices.

Symmetric matrices have the advantage that only half the off-diagonal elements need to be stored, reducing the overall storage by nearly 50%. It also presents a new constraint for us to add to constraints 3 and 4. For COO formatted matrices, assuming that the non-zero elements are stored in the same fashion as those of a lower-triangular matrix, we have a constraint involving both the x and y index arrays:

$$x[k] \geq y[k] \tag{6}$$

where $1 < k < \mathrm{NNZ}$. (This constraint does not have much meaning when applied to CSR formatted matrices, since x_{off} represents an offset within the other two arrays, rather than a fixed index value.)

There are a few advantages of storing only half the off-diagonal elements of symmetric matrices. First, there is the obvious savings in space of roughly one half. Secondly, the probability of a bit-flip affecting an element's index or value is also approximately halved, since the matrix only occupies half as much space. Finally, the use of constraint 6 often reduces the number of susceptible bits in a given index by at least 1. For example, we see this if we have a non-zero element on a diagonal with at least one valid index (along the minor dimension) between it and the next non-zero element. In this case, at the times when the least significant bit is zero in this index we now have additional protection for the least significant bit. Similarly, if any of the most significant zero bits could have been changed (if the matrix had been stored fully, as with an unsymmetric matrix) and still yield a valid index, these bits too would now be protected by constraint 6.

Exploiting Sparse Matrix Banding Constraints

Banded matrices are another common type in the Florida collection. Banded matrices comprise of regions of non-zero elements that are grouped along diagonals, often the main diagonal. These regions of the matrices, if viewed alone, appear densely packed. Banded matrices can also be exhibited through blocks of diagonals, yielding (banded) block-diagonal matrices. In these sparse matrix structures, values of index positions that are between the extremities of the band (the *left* and *right bandwidths*) have little freedom for a bit-flip to affect their index data yet remain bound within the band.

When looking at banded matrices, constraints 3 and 4 can detect bit-flip errors in a majority of the bits of the indices, with only errors in the least significant index bits potentially going undetected. The more densely packed the elements within the band, the greater the number of index bits that will be protected by our scheme. With such close non-zero index grouping, the more chance an index has of being correctable if and when a detectable bit-flip occurs. For bit-flips that occur in the most significant index bits for elements within the bandwidth (a tighter bound than the dimension), there is a higher probability of successful correction since there is a higher probability that there is only 1 bit of the index which could be re-flipped in order to satisfy all the constraints. Conversely, the more sparsely distributed the elements within the band, the less the protection the constraints can offer.

Considering specific classes of banded sparse matrices, we can develop tighter bounds for the ordering constraints. In a single-banded matrix, or one that is close to it in structure, the number of valid index positions within the band will likely be small, typically under 1% of the appropriate dimension. This structure results in the non-zeros being very close together within the bands, with subsequently close indices. For sparse matrices in the Florida collection, which are all of dimension up to 2^{27}, this would mean that a bit-flip to any of the most significant 7 bits of the index (top 12 bits of the whole 32-bit number) would immediately violate a constraint, and thus over a third (37.5%) of all potential single bit-flip errors affecting the indices could be corrected by this single constraint alone. The smaller the bandwidths, the tighter the effect of the constraints.

In this way, bit-flips to the guarded index values are limited in the damage they can cause, with spatially closer indices better protected against bit-flips than those spread further apart. If the non-zero elements are consecutively spaced within the bands, the danger of bit flips in the indices is removed almost entirely; single bit-flips in those elements not at either bandwidth extremity would violate one of constraints 3 or 4 and be both detectable and correctable, whilst those occurring in the few elements at the band extremities would have an improved chance of detectability and possibly correctability. For non-zero elements that are more spread out, the danger of a 'silent' bit-flip in an index which creates another legitimate index that retains the same ordering is increased. In this case, the index bits — especially the least significant ones — are more vulnerable and may require further protection.

The discussion above can be generalized to matrices with more than one band. The greater the distance between the bands, the more likelihood there is of bit-flips to the indices of the band extremities changing into another valid index, and therefore causing a change in an index that our constraints would not detect. However, the more bands there are, the smaller these distances may be, so despite matrices with a greater number of bands containing a greater number of susceptible elements (those on the band extremities), the

smaller the chances are that these susceptible elements are affected by bit-flips without violating constraints. In the vast majority of cases the number of non-zero elements at the band extremities is a small proportion of the total number of non zeros, and therefore banded sparse matrices should benefit even more from our scheme than non-banded ones.

Efficacy of the Sparse Matrix Constraints Scheme

In the following section we detail the results from applying all of the COO-applicable constraints detailed above (Equations 1 – 4). These constraints were applied to all of the real matrices within the University of Florida sparse matrix collection – roughly ∼60% of the 2,600 example matrices. We found that, as expected, matrices exhibited different levels of amenability to our constraint-based fault tolerance scheme, with some benefiting extremely well and others receiving much less protection. In this section, we present results for the number of protected bits (Pb) in the indices from three exemplar sparse matrices chosen to illustrate the benefits our scheme can deliver across a range of different matrix sizes and types (small, medium and large). These examples were also chosen as being representative of the results we observed across the whole collection.

In each of the three cases, we provide two graphs. The first of each pair is a graph which shows the percentage of bits in each index which are protected by applying constraints 1 – 4 from our scheme. If our scheme were to fully protect all bits in every index, this graph would show 100% across the full range of index bits on the x axis. If our scheme protected only the least significant 16 bits in every index, it would show 0% for 32-17 bits and then 100% for 16-1 bits. If half the indices had every index bit fully protected and the other half had no index bits protected at all, then the graph would show 50% right across the x axis. The second graph in each pair is a histogram based on the bit specific positions in each index which remain unprotected by our scheme. This graph takes into account the susceptible bit positions of each of the indices in the matrix and, assuming that a bitflip error occurs, plots the expected frequency (or likelihood) in which each susceptible bit position would be altered on the y axis against the susceptible bit positions in the x axis. A bin showing 15% for bit position 1 means that across all the susceptible bit positions in all of the indices there is a 15% chance that an occuring bitflip would alter the least significant bit of an index. The more skewed the graph is, the greater the likelihood that bitflips will affect specific bits. If our scheme worked perfectly one would see all bit positions showing 0%.

The first pair of graphs are from a small, unsymmetric, rectangular sparse matrix, named lp_agg in the University of Florida collection. This matrix is relatively small compared to the rest of the collection, being just 488×615 in size and containing 2,862 non zero elements (a 0.95% fill rate). Due to its relatively small size, there is very little clustering of this matrix's non-zeros, and therefore many relatively large gaps between elements. Fig. 2a shows that, whilst all elements have at least 22 bits of each index fully protected from our first set of constraints, the other constraints in our scheme make less of a difference in this scenario. Even with this very small size our scheme has protected over two thirds of the index bits, and from Fig. 2b, one can see that the index bits that remain unprotected are almost all in the least significant 9 bits of the indices. Even though our scheme has worked well in this case, a matrix as small as this is not really a candidate for an Exascale-class computation.

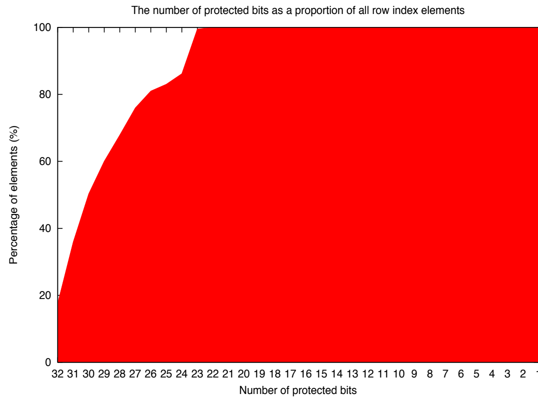
The second pair of graphs are from an average sized, symmetric, square sparse matrix, named nasasrb. This matrix is of dimension $54,870 \times 54,870$ elements, and contains 2,677,324 non zeros (a 0.089% fill rate). Here we see an excellent result for our scheme. Fig. 2c shows that the first set of constraints fully protect 16 bits of every index, with the remaining constraints fully protecting an additional 4 bits, as well as ensuring nearly 70% of all elements are fully protected. One of the reasons we achieve such a strong result in this case is that this matrix's symmetry enables additional constraints and thus protection. Very few index bits are left unprotected in matrices of this class. Fig. 2d shows that the majority of the index bits that remain unprotected are in the least significant byte of the index.

The final pair of graphs are from a large, unsymmetric, square sparse matrix, named circuit5M. This is one of the largest real matrices in the collection, being $5,558,326 \times 5,558,326$ elements in size and containing 59,524,291 non zeros (a 0.00019% fill rate). Our scheme also works well with this matrix. Fig. 2e that we are able to fully protect all 32 bits of around 43% of all indices, and in the worst case any one index has at least 10 bits protected by our scheme. Fig. 2f shows a fairly even spread of index bits which remain unprotected, with each susceptible bit position almost as likely as any other (around a 5% likelihood). The fraction of index bits left unprotected then drops off sharply, and by bit position 24 all the remaining index bits are fully protected in every index.

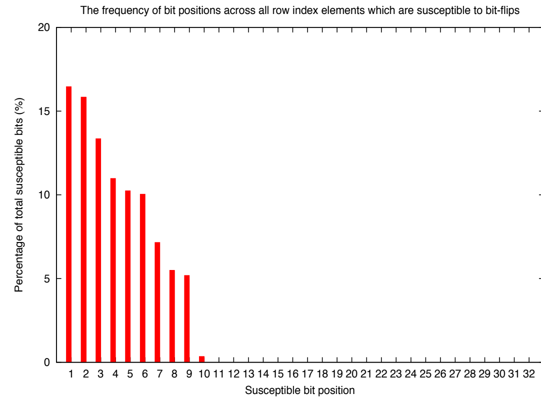
Software-based Error-Correcting Codes

The criteria-based fault tolerance techniques presented in the previous sections are a low overhead way of catching many, but not all, bit-flip faults in the indices of sparse matrices. The techniques have two disadvantages. First, as illustrated in the results, not all bit flips in the indices can be caught by the criteria checking, with some bit flips generating incorrect indices that will slip through. Second, the criteria checking will only protect the indices, and not the floating point data values. In order to address these shortcomings, we will now present some additional techniques which have the potential to provide full protection for both the indices and the data values, from both single and double bit errors. These additional techniques exploit *software-based error correcting codes*.

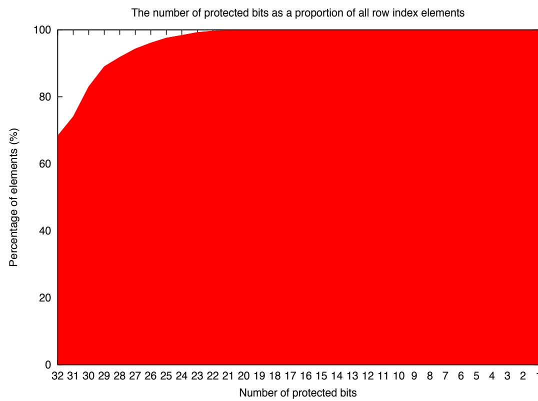
Error-correcting code, or ECC, is an approach based on techniques originally developed by Hamming (19). ECC can protect data from bit-flips by adding an extra set of bits to the data in such a way that a change to any single bit of the data (or of the additional bits) can be detected — and with an appropriate scheme, corrected. It does this by encoding the additional set of bits, called the *parity vector*, with the bits to be protected, the *data vector*, into a mixed sequence, the *encoded vector*. Decoding this encoded vector yields the *syndrome vector* (of identical length to the parity vector). A



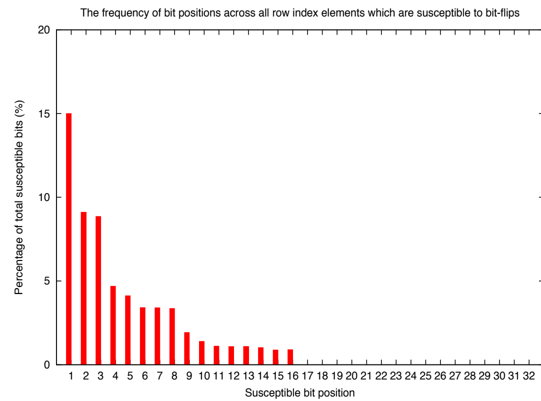
(a) Pb graph for a small, unsymmetric matrix (lp_agg).



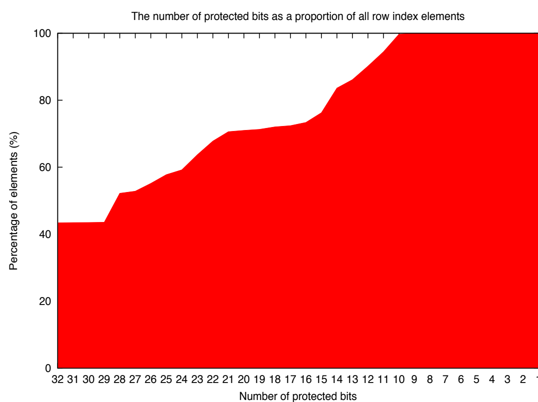
(b) Index bit positions (lp_agg).



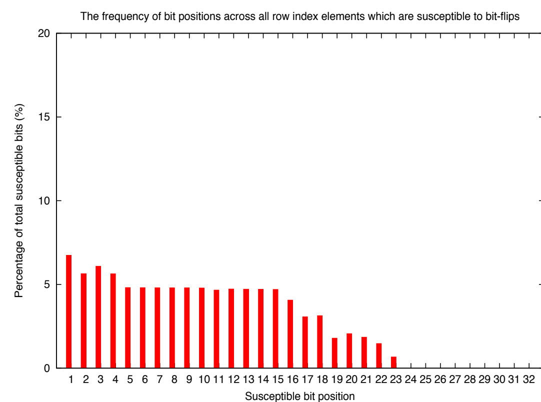
(c) Pb graph for a mid-sized, symmetric matrix (nasasrb).



(d) Index bit positions graph (nasasrb).



(e) Pb graph for a large, unsymmetric matrix (circuit5M).



(f) Index bit positions (circuit5M).

Figure 2. Graphs for three representative (real) matrices of different sizes from the University of Florida collection.

non-zero syndrome vector indicates that an error has taken place, and its value indicates the position of the bit that was altered (20). The number of additional bits required for the parity vector can be determined by noting that the number of positions which can be indexed by the parity bits — 2^p for p parity bits — must be at least the total of the number of data bits, d, summed with the number of parity bits, p, plus 1 extra bit to indicate error-free conditions, i.e.,

$$2^p \geq d + p + 1. \tag{7}$$

This means that, for example, 4 parity bits can protect at most 11 data bits, whilst 8 parity bits can protect at most 247 data bits. The above explanation holds true when there is only a single bit-flip that occurs. Another possible,

though much less frequent scenario, is a double-bit error in the same data word. Double bit errors are fifty times less likely to occur than single-bit errors, themselves a very rare occurrence (11). Single error correct (SEC) methods will fail if double-bit errors occur, and a different approach is needed to at least detect a double bit-flip. A variant of this scheme is single error detect (SED), which, as the name suggests, detects single bit errors without correcting them. SED schemes are amongst the simplest possible as they only require 1-bit parity checks.

Determining the locations of two bit-flips is much harder than the single bit-flip case, and requires significant additional overhead. As a result, instead of employing a technique to correct two bit-flips, it is much simpler to add one extra bit to whatever parity vector length has been chosen and use that extra bit as a parity check on all the other bits. This extra bit must be independent of all the other bits. With this approach, single errors can be both detected and corrected, whilst double errors can be just detected, giving rise to the name single error correct, double error detect (SECDED). A common SECDED scheme uses 8 parity bits, with one bit to detect double errors and the remaining seven bits used to protect up to 120 data bits.

This ECC encoding technique can be modified for data such that, instead of having a separate *additional* parity vector, mixed with the data vector to get the syndrome vector, the parity vector is *embedded* in 'repurposable' bits from some combination of the sparse matrix element indices and data values. The constraints developed in earlier sections offer an initial insight into which bits of the element indices may be repurposable; instead of using the upper unused region of index bits to detect a violation of the first set of constraints, these bits could instead be used to store parity data. These repurposed bits could then be masked off when the actual index value is required.

Repurposing Bits to Achieve ECC-like Fault Tolerance

For a given sparse matrix, for any matrix smaller than the maximum representable unsigned 32-bit integer ($2^{32} - 1$), there will be some number of bits at the most significant end of the indices which are not being used. These unused bits offer a potential location in which parity bits could be stored to provide an ECC-like scheme for index protection. These repurposable bits are illustrated in Figure 3. Notice that by repurposing existing bits, rather than using additional memory, as in existing hardware-based ECC schemes, we will typically save 12.5% memory capacity and memory bandwidth (typical ECC hardware schemes add 8 parity bits to protect each 64 data bits).

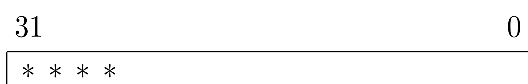


Figure 3. The most significant index bits that are not used (marked with asterisks) could be repurposed.

An analogous observation can be made for the least significant bits of the fraction of a double-precision floating-point number, since they represent a relatively insignificant part of the value contained within. However, protecting all 64-bits of a double precision floating-point value by only using bits within the 64-bit value itself, would require the 7 least significant bits of the fraction for the embedded parity vector: six bits to index all the bits of data, along with an extra bit to allow for error-free syndromes. This would introduce 7 bits of 'noise' into the floating-point data, which in the worst case would tend towards a maximum relative error in the value's fraction of $1/2^{(52-(7+1))}$, or $5.68 \times 10^{-12}\%$. The noise may be acceptable for iterative sparse matrix calculations, but there is of course the risk that the more noise present, the more likely it becomes for an iterative solver using data protected through the scheme to take longer to converge, or in the worst case, to fail to converge all together. The effect of noise caused by bit-flips in floating-point data on the convergence of iterative solvers has been analysed in work by Elliott et al in (16). They showed that a stationary iterative solver successfully converged in spite of single bit-flip errors in the arithmetic, and without requiring additional iterations.

Rather than considering the indices and data values as separate items to be protected with ECC individually, it is helpful to consider each sparse matrix element as a compound data item, containing both the indices and the data value at that index. In the case of COO format sparse matrices, this results in a 128-bit compound element: two 32-bit unsigned indices, and one 64-bit floating point value. By considering a 128-bit compound element in this way, we can derive a SECDED scheme while minimizing the number of bits used for parity checks and simplifying the resulting syndrome vector. This scheme would only require 8 parity bits *in total*. If alternatively we had tried to protect each index and data component in isolation, we would have required 6 bits for each 32-bit index and 7 bits for the 64-bit floating-point value, or 19 bits in total. Thus by treating these three components as a single compound element, the number of parity bits required for individual ECC schemes is more than halved.

A scheme that considers a 128-bit sparse matrix compound element as a single entity has an additional benefit, in that donor bits that make up the new parity vector can be carefully chosen from different parts of the indices or data value. Different distributions of donor bits between the indices and the floating-point value will result in different tradeoffs between the maximum dimension supported by the indices (and thus maximum matrix size) and the amount of noise introduced into the least significant fractional bits of the value. There is one limitation which must always be obeyed when placing donor bits within the 128-bit compound element: parity bits must be placed in positions that are mutually independent.

Table 3. The positions of the parity bits in a simple powers-of-2 positioning scheme, numbering from 1 (least significant) to 128 (most significant).

Parity bit:	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
Bit position:	1	2	4	8	16	32	64	128

The simplest placements would be in power-of-2 positions, as shown in Table 3, which would give full independence — each parity bit could be calculated in parallel. However, if each of the parity bits were placed in

these positions, the scattering would make data extraction overly complex. This obstacle can be removed by storing the parity bits in one location and treating them as if they were in another for purposes of parity calculation. In this manner, some or all of the parity bits can be moved to another position within the 128-bit compound element, provided that there is some mechanism (implicit or explicit) that uniquely determines which position they represent. For example, the eighth parity bit (the DED bit) may be stored in the most significant bit of the first index ('position 1') instead of the last bit of the data element ('position 128'). When parity calculations and checks are performed, it is calculated as if it were in the last position of the 128-bit compound element; either the bit must be moved prior to calculation or there must be some method for performing calculations that refer to it as though it were in bit position 128.

Five of the most efficient distribution layouts of parity bits are listed in Table 4. In the first and last cases, the parity bits are stored in a single byte for efficient access. In the middle three cases (as well as the last case), there is an even split in the number of parity bits placed in the upper bits of each index to allow for maximal storage of matrices which have the same dimension for both rows and columns.

In the first row of Table 4, all eight parity bits are stored in the most significant byte of the first index element. In the second row, the eight parity bits are split equally across the two index elements. In the last row, all eight parity bits are stored in the least significant byte of the floating-point value. Each row in between represents a trade-off between limiting the range of the indices and the precision of the floating-point value. The second row (using the top four bits of each index element) is the most efficient layout for square matrices which also does not affect the precision of the floating-point value. In the first row, where all eight bits are stored in the first index element, a square matrix would not be able to use the upper eight bits of the second index element, and thus the maximum matrix size that this choice would work for would be $2^{24} - 1$ squared.

Parity checks are performed on the 128-bit compound element as a whole, in a near identical way to how they would be performed by conventional ECC. A series of pre-determined bitmasks can be used to ensure parity checks are performed on the elements as they would appear if the layout had the parity bits in powers-of-2 positions. Together with a check on the DED bit, this allows for a full SECDED scheme to be implemented in software using only the 128-bit compound element itself and the relevant additional logic – no additional memory or memory traffic is required, as there is with traditional ECC hardware schemes. The operations required for encoding, decoding, and extracting the compound element data are all elementary (simple bit masks) and efficient to implement in software on most computer architectures.

To calculate the floating-point value in the bottom three cases in Table 4 either the lowest bits could be zeroed out or the parity bits could be left as the least significant bits in the floating-point number's fraction, and thus treated as noise. It is also worth noting that the values of the parity bits do not change (unless they encounter bit-flips themselves) throughout the calculation, and as such the amount of noise present can be calculated before an application using the sparse matrix data is run.

Application to CSR/CSC Formats The technique for finding 8 bits for a parity vector across an entire 128-bit compound element has only been described here in the context of the COO sparse matrix scheme. COO is a simple format, with an i, j index per 64-bit value. The scheme can also be applied to the alternative CSR format, by observing that all the scheme requires of the data that makes up the 128-bit compound element is that it includes two 32-bit quantities with 'headroom' and one 64-bit quantity (that may or may not have 'legroom'). A simple scheme to apply this approach to CSR format sparse matrices would be to only use bits from the larger row or column index array. This would impose a larger restriction in the size of the sparse matrix, since it would steal all 8 required bits from one index, and thus limit the maximum size of the array in that dimension to 2^{24}. However, this scheme would be more than adequate for the vast majority of sparse matrices in the real world, and its simplicity would mean that its overheads would be low compared to other approaches.

Performance overheads for software fault tolerance schemes

Our software ECC scheme can protect a sparse matrix just as well as a hardware ECC scheme would. Depending on the exact scheme chosen, we can implement either single or double error detect, with further options for correction. These different schemes will all carry different amounts of run-time overhead, and so in this section we present performance results for several different schemes.

We developed a simple sparse iterative CG solver. Our experimental platform was the University of Bristol's Blue Crystal Phase 3 supercomputer. Each node has dual socket Intel Xeon E5-2670 CPUs, with a total of 16 x86 cores running at 2.6 GHz and 64GB of memory. The nodes run Red Hat Enterprise Linux 6.4, and Intel's compiler v16 with compiler flags "-O3 -march=native -xHost -restrict -fp-model fast -static-intel -ipo". All tests were run five times with the average time taken.

Several different schemes were developed and bench-marked, which measured the overheads of the full range of index criteria checking and ECC schemes we have presented. Overheads were measured for a set of four matrices with different sizes and spread of non-zeros, taken from the University of Florida collection. These were, listing from smallest to largest: Pres_Poisson ($54,870^2$, 2,677,324 non-zeros), nasasrb ($147,900^2$, 3,489,300 non-zeros), parabolic_fem ($525,825^2$, 3,674,625 non-zeros) and thermal2 ($1,228,045^2$, 8,580,313 non-zeros). Across these matrices the number of non-zeros was in the range 0.001% to 0.1%. The CG solver was set to iterate until it had converged with a tolerance of 10^{-6}.

The results are shown in Figure 4. There are a number of interesting observations to make about the observed overheads. First, the relative cost of each of the schemes is highly problem dependent, with a general trend for the costs to be smaller the larger the sparse matrix. Second, the single bit detect only (SED) scheme is the cheapest overall. Its overheads are in the range 0.5% to 28.5%, with the smaller

Table 4. Parity bit placement choices and costs for a 128-bit compound element.

Parity bits (index-index-data)	Split ordering of 128-bit compound element; each 'a-b' corresponds to the parity-index/data-parity split	Extra operations for index extractions
8-0-0	(8-24, 0-32, 64-0)	1x simple mask
4-4-0	(4-28, 4-28, 64-0)	2x simple mask
2-2-4	(2-30, 2-30, 60-4)	2x simple mask
1-1-6	(1-31, 1-31, 58-6)	2x simple mask
0-0-8	(0-32, 0-32, 56-8)	none

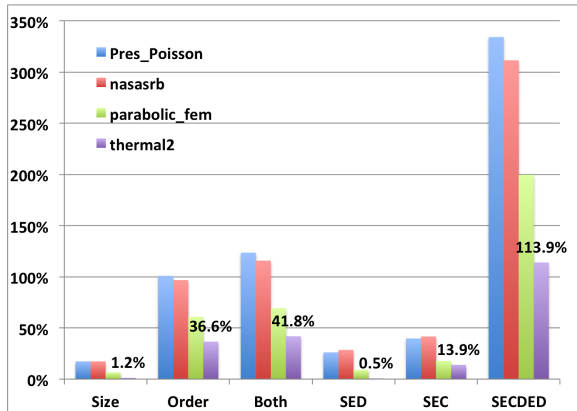


Figure 4. Overheads for each of the fault tolerance schemes. The first three groups of bars are for the constraint checking schemes, while the last three are for the software ECC schemes. Data labels are included for the largest sparse matrix, "thermal2". Overheads are given as a relative overhead compared to the baseline CG solver.

overheads also on the largest matrices. This is an interesting result as it shows that a software-based single bit flip detect scheme can be implemented with negligible overhead on large enough problem sizes. The perhaps surprising result though is that this scheme has a lower overhead than that of applying the index constraints; applying both size and ordering constraints for the same large "thermal2" sparse matrix required a 41.8% overhead, compared to the 0.5% overhead to perform a full single error detect on the complete 128-bits of the compound sparse matrix element. Thus the SED scheme is providing more complete protection at a much lower overhead in this instance.

Considering the overheads of providing not just detection but also correction, we see that single error correct (SEC) is not much more expensive to implement than just SED, with overheads for SEC ranging from 13.9% to 41.6%, compared to the 0.5% to 28.5% for SED. However, extending this scheme to also detect (but not correct) double bit errors is very expensive. Overheads increase significantly and are in the range 113.9% up to 334.1%. We have not found a way to make SECDED significantly faster in software, and so for now this scheme appears to be too expensive to be practical. However SED or SEC look relatively cheap to implement for large enough matrices, which are in any case those matrices which will need to be protected from silent data corruptions. For the large "thermal2" sparse matrix, SED and SEC require just 0.5% and 13.9% overhead respectively, both of which are below the 20% performance overhead that traditional checkpoint/restart is regarded to incur. If one were to also reclaim the 12.5% storage and bandwidth overhead required by ECC hardware schemes, then a single error detect or correct scheme in software could have a significant advantage over the current state of the art.

Discussion and Conclusions

The work presented in this paper has focused on developing software-based fault tolerance techniques for detecting and, in the case of our ECC schemes, correcting bit-flip errors in sparse matrices.

The overheads of our constraint checking scheme are perhaps higher than one would have expected, incurring a 41.8% penalty even on the largest sparse matrix in our test (thermal2). Thus even though the scheme provides a good level of protection for the indices, this approach alone does not appear to be practical. This kind of criteria checking is potentially a candidate for acceleration in hardware though, and instruction set extensions might bring this overhead down considerably.

Our software-based ECC scheme was much more successful, providing full single error detect protection for both the indices and the data in the sparse matrix. Overheads were also acceptable, with SED costing just 0.5% and SEC just 13.9% on the largest sparse matrix (thermal2). However, at this time, a full SECDED implementation in software appears to be too expensive to be practical. This might be another area where innovations in instruction set design might yield significant benefits.

For both of these approaches (index criteria checking and software ECC), we have been aiming for overheads of less than the alternate checkpoint-restart scheme. These are widely predicted to be increasingly costly for Exascale machines, potentially requiring a performance overhead of $\sim 20\%$ or more. The performance of our best scheme, the software ECC single bit techniques, is potentially much better than the current combination of hardware ECC and checkpoint-restart. We have an additional benefit in the presence of faults: in our scheme, a user-level software routine can discover exactly which sparse matrix element has become corrupted and can restore just this one object. In contrast, a traditional checkpoint-restart scheme involves heavyweight OS-level intervention and potentially a large amount of data transfer from disk across the network.

The potential energy savings of our software-based fault tolerance scheme had yet to be quantified, but we can already show that our scheme reduces the amount of data that needs to be moved to support ECC and instead performs an increased number of simple arithmetic operations (integer arithmetic, compares and conditional tests). As data movement is expected to be energy expensive

compared to integer operations, this is the right trade-off to make in the bid to improve energy efficiency for Exascale.

To conclude, when working with sparse matrices whose elements are stored in a consistent order, we have shown that it is possible to define schemes to detect and potentially correct bit-flip errors from affecting the underlying index bits. We have also shown that a low-overhead software ECC scheme can protect the entire sparse matrix, potentially saving memory bandwidth and capacity compared to hardware ECC, while reducing the need for traditional checkpoint-restart and its associated heavy demands on system-wide resources. The Application-Based Fault Tolerance (ABFT) techniques we have presented in this paper could be added to existing sparse matrix solver libraries, and would help improve their built-in fault tolerance as a step towards making Exascale-class computations a reality.

Acknowledgments

The authors would like to thank EPSRC and NAG Ltd for their funding of the research undertaken in this study. This work was also supported by funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the Mont-blanc 2 Project (www.montblanc-project.eu), grant agreement n° 610402. This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol, including the Blue Crystal Phase 3 supercomputer – http://www.bris.ac.uk/acrc/.

References

[1] Ziegler JF, Lanford WA. Effect of cosmic rays on computer memories. Science. 1979 November;206(4420):776–788.

[2] Ziegler JF. Terrestrial cosmic rays. IBM Journal of Research & Development. 1996 January;40(1):19–39.

[3] Ziegler JF, Curtis HW, et al. IBM experiments in soft fails in computer electronics. IBM Journal of Research & Development. 1996 January;40(1):3–18.

[4] Ziegler JF, Lanford WA. The effect of sea level cosmic rays on electronic devices. Journal of Applied Physics. 1981 June;52(6):4305–4312.

[5] Borucki L, Schindlbeck G, Slayman C. Comparison of accelerated DRAM soft error rates measured at component and system level. In: IEEE International Reliability Physics Symposium; 2008. .

[6] McKee WR, McAdams HP. Cosmic ray neutron induced upsets as a major contributor to the soft error rate of current and future generation DRAMs. In: IEEE International Reliability Physics Symposium; 1996. .

[7] Fang YP, Vaidyanathan B, Oates AS. Soft error rate cross-technology prediction on embedded DRAM. In: IEEE International Reliability Physics Symposium; 2009. .

[8] Hwang AA, Stefanovici IA, Schroeder B. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XVII. New York, NY, USA: ACM; 2012. p. 111–122. Available from: http://doi.acm.org/10.1145/2150976.2150989.

[9] Schroeder B, Pinheiro E, Weber WD. DRAM errors in the wild: a large-scale field study. In: ACM International Joint Conference on Measurement and Modeling of Computer Systems; 2009. .

[10] Dongarra J, Beckman P, Moore T, Aerts P, Aloisio G, Andre JC, et al. The international exascale software project roadmap. International Journal of High Performance Computing Applications. 2011;25(1):3–60.

[11] IBM. Fault tolerance decision in DRAM applications. IBM; 1997.

[12] Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, et al. Exascale computing study: Technology challenges in achieving exascale systems. Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech Rep. 2008;.

[13] Hoemmen M, Heroux MA. Fault-Tolerant Iterative Methods via Selective Reliability. Sandia National Laboratories, Albuquerque, NM 87175: Sandia National Laboratories; 2011. Available from: http://www.sandia.gov/~maherou/docs/FTGMRES.pdf.

[14] Davis TA, Hu Y. The University of Florida sparse matrix collection. ACM Transactions on Mathematical Software. 2011 Dec;38(1):1:1–1:25.

[15] Hukerikar S, Diniz PC, Lucas RF. A Programming Model for Resilience in Extreme Scale Computing. In: Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on; 2012. .

[16] Elliott J, Mueller F, Stoyanov M, Webster C. Quantifying the Impact of Single Bit Flips on Floating Point Arithmetic. One Bethel Valley Road, Oak Ridge, Tennessee 37831: Oak Ridge National Laboratory; 2013. ORNL/TM-2013/282. Available from: http://info.ornl.gov/sites/publications/files/Pub44838.pdf.

[17] Maruyama N, Nukada A, Matsuoka S. A high-performance fault-tolerant software framework for memory on commodity GPUs. In: IEEE International Symposium on Parallel & Distributed Processing; 2010. p. 1–12.

[18] Saad Y. Iterative Methods for Sparse Linear Systems. SIAM; 2003.

[19] Hamming RW. Error Detecting and Error Correcting Codes. The Bell System Technical Journal. 1950 April;29(2):147–160.

[20] Warren HS. Hacker's Delight. Addison-Wesley; 2012.