LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-618412

Comprehensive Algorithmic Resilience for Numeric Applications

S. Chen, G. Bronevetsky, M. Casas-Guix, L. Peng

February 13, 2013

International Conference on Supercomputing
Eugene, OR, United States
June 10, 2013 through June 14, 2013

Disclaimer

Comprehensive Algorithmic Resilience for Numeric Applications

ABSTRACT

As High-Performance Computing (HPC) systems become larger and more complex and the feature sizes of their electronic components grow smaller, the systems become increasingly vulnerable to soft faults. Since such faults may cause application crashes or may silently corrupt application results it is necessary to develop techniques to protect applications from such errors at a low cost in performance and power. Despite extensive work on general-purpose resilience techniques, they are cost effective for protecting only certain system components, such as memories. To provide comprehensive protection for the computations of HPC applications it is necessary to develop algorithmic resilience mechanisms that use the properties of algorithms to provide cost-effective protection.

Algorithmic resilience can be very complex to deploy in applications because a comprehensive resilience strategy must address all the types of errors that an application may experience. This includes corruptions to numerical data as well as control information and data structures. Full protection requires the combination of multiple techniques including algorithmic error detection, crash detection, replication of key control information as well as support for restart when errors are detected. Further, it is necessary to configure the properties of the resilience mechanisms to make sure that they are cost-effectively meeting the user's accuracy and confidence targets. This paper shows how all these issues can be resolved comprehensively in the context of three numeric applications.

1. INTRODUCTION

The increasing size and complexity of HPC systems is making them increasingly vulnerable to soft faults. Soft faults are transient corruptions of circuit state caused by physical phenomena such as strikes by neutrons or alpha particles [1, 18] or thermal electrical noise [15]. They can affect the state of processor latches and registers and may cause the application to crash or worse, to silently return incorrect results [4]. As the feature sizes of electronic circuits shrink, technology scaling will make soft errors a larger problem [17] due to the fact that each circuit element will hold less charge and can thus be disrupted more easily. These phenomena make it imperative to develop techniques to make HPC systems resilient to soft faults.

Resilience is a problem that must be addressed at all levels. While materials science and circuit design techniques can be used to improve resilience, they come at a cost in reduced power efficiency and performance that can become prohibitive if processors need to be sufficiently reliable to build a large HPC system. Techniques such as error correcting codes have been very effective at making memories and caches resilient to soft faults [16]. However, they are more expensive for protecting core-internal state such as latches and are significantly less effective for checking the correctness of computations. Traditional approaches like cross-core or node replication [7, 13] are expensive in their use of computing resources and power and their long error detection latencies require processor state to be frequently checkpointed. Processor designs that incorporate instruction replication [21] offer fine-grained error detection and rollback but require more power as well as novel hardware features that are not likely to be included in the commodity processors used in HPC systems for cost reduction reasons.

The limitations of automated resilience techniques has motivated the need for algorithmic techniques that can detect and tolerate errors by taking advantage of application-specific properties and invariants [3, 24, 11]. Although these techniques can be very efficient, most research work on such techniques has primarily focused on how their algorithmic invariants can be leveraged to enable accurate error detection. While this goal is important, it ignores the full complexity of the problem since it focuses on only a subset of the errors and does not consider combinations of different resilience techniques to efficiently and effectively address all the different types of faults. In reality a comprehensive solution must incorporate a range of additional techniques such the replication of key variables, detection of memory violations and localized rollback on detected errors, as well as techniques to configure these various resilience methods to achieve the best performance given the user's accuracy and confidence targets.

This paper presents a comprehensive algorithmic resilience approach that considers all of the above issues. We demonstrate it on different types of numerical applications: the Alternating Direction Method of Multipliers (ADDR) for Lasso problems [2], the Hattrick gravitational simulation [20] and

the Digital Room Correction (DRC) acoustic correction application [22]. We show how these three applications can be comprehensively protected from soft faults by adding three different resilience mechanisms to the routines where they spend the most time: algorithmic error checks, replication of critical data structures and checkpoint-restart of individual routines. We demonstrate via fault injection experiments that although none of these techniques can individually protect applications from soft faults, their combination is highly effective. Further, we demonstrate how these techniques can be configured to offer the right level of protection at any given fault rate or input type. More specifically, these configurations can be used to produce a wide range of overheads and result accuracy levels and to tune the probabilistic confidence level that the given overhead or accuracy target will be met.

The paper is organized as follows. Section 2 discusses the problem of soft errors and presents the fault injection model we use to simulate them in our experiments. The applications to which we apply our resilience methodology are described in Section 3. Section 4 summarizes the major routines of these applications and describes how these resilience techniques are applied to them. Further, it experimentally evaluates the resilience techniques, describing their overheads and the accuracy levels of the protected routines at a wide range of possible error rates. The effectiveness of our methodology for protecting the full applications is evaluated in Section 5.

2. SOFT FAULTS

Cosmic rays or alpha particles hitting the computer circuits and accumulating a sufficient amount of energy may invert the device logic states from "0" to "1" or from "1" to "0" [25, 1]. These inversions are termed as soft errors or soft faults. The shrinking processor and memory feature size and the lower threshold voltage coming with relentless manufacturing technology scaling make modern high performance computers are highly vulnerable to soft errors which propagate as the program is being executed. Prior studies have demonstrated that soft errors occurring in all the important bits that are required for Architecturally Correct Execution (ACE) [19] eventually change the program outcome, i.e., manifest errors to the application. The ACE bits include critical information to keep the program logic such as the program counter register for a committed instruction. In the past decades, soft errors have brought significant losses on commercial servers and high performance computers such as the recall of Sun servers due to memory errors [1] or the cache corruptions in the ASCI Q system [18].

We use a fault injection infrastructure [5] that simulates the architecture-level effects of soft faults. It works by compiling application into the Low Level Virtual Machine (LLVM) typed bytecode [14]. This bytecode is transformed to allow a bit flip in each instruction's output with some probability P. The time between injections is thus exponentially distributed with period $\frac{1}{P}$. Our fault injector is a high-level approximation of the physical corruptions caused by soft errors that strikes a balance between model accuracy and cost, as do related approaches [10, 19]. Alternative models, such as injection into gate-level processor models [6] or neutron beam irradiation [12] are possible but their significantly higher cost makes them impractical for conducting large-scale fault injection campaigns such as the ones reported in this paper.

3. TARGET APPLICATIONS

We evaluate our resilience methodology by applying it to the following three applications. Since they spend most of their execution time in library routines, we focus our resilience efforts on protecting these routines, which are described in Section 4.1.1.

3.1 Alternating Direction Method of Multipliers (ADDR)

The ADDR method solves under-unconstrained linear problems $Ax = b$ for x (A has fewer rows than columns) while minimizing the function $f(x)$. This is an iterative algorithm uses 64-bit precision and that spends most of its time in the following linear algebra operations, matrix-matrix multiplication, matrix-vector multiplication, rank-k update and Cholesky factorization, using the implementations of these routines from the GNU Scientific Library [8]. We focus on a specific application that uses the ADDR method to solve Lasso problems, which are defined as minimizing the function $\frac{1}{2}\,||Ax-b|\,|_2^2+\lambda*||x|\,|_1$. It is parallelized using MPI and our experiments focus on 4-processors runs of ADDR which computes linear systems of size $\{800, 1200, ..., 3600\} \times 500$ (800 to 3600 measurements with 500 variables each) and in the experiments below are denoted $S800$, $S1200$, ..., $S3600$. The values in A and b are generated by sampling a normal distribution with a mean of 0. According to our experiments, the running time of this solver is not dependent on the kind of distribution the inputs are sampled from.

3.2 DRC: Digital Room Correction

DRC is a sequential program generates filters for HiFi audio systems to compensate for the reflection of sounds in a room, using impulse response measurements of the audio equipment and considering the positions of the listeners. The inputs are stored in Pulse Code Modulation (PCM) format, which is an array of 32-bit floating point numbers representing the samples at each sampling time step and the computations are performed using 32-bit precision. Although the algorithm is divided into many phases the most computationally intensive step consists of the GSL's implementation of the Fast Fourier Transform and a DRC-internal implementation of Finite Impulse Response filter generation. This program accepts user-controllable configurations to decide the width and number of iterations of FFT transforms it uses to perform various passes on the input file, such as windowing, deconvolution, pre- and post-filtering. The input used in this paper is an audio file of size 768 kilobytes, which is sampled at the rate of 30Khz, 40Khz, ..., 90Khz.

3.3 Hattrick

Hattrick is a sequential application simulates the motion of bodies under the effects of gravity using 64-bit precision. It is designed to help discover extra-solar planets by inferring their existence from Transit Timing Variations, where the difference between the times when a planet passes in front of its host star is used to infer the existence and properties of other planets in the system. This program spends most of its execution time in the GSL solver for Ordinary Differential Equations to solve the system's equations of motion. A given input is described using three parameters: P=number

of planets, T=amount of time to simulate, and A=the accuracy target. In our experiments we considered the following four inputs: $P2T2090A15$, $P2T3090A15$, $P2T4090A15$ and $P3T2090A11$, where an $A15$ and $A11$ denote accuracy targets of $1e-15$ and $1e-11$, respectively.

4. RESILIENCE TECHNIQUES

In this paper we consider a comprehensive resilience strategy that combines various types of error detection and recovery. Error detection techniques are discussed in Section 4.1, with Section 4.1.1 focusing on algorithmic techniques to detect errors in the numeric portions of application state, while Section 4.1.2 focuses on protection of data structures. Section 4.2 then considers recovery techniques based on both checkpoint-restart, which offers lower-overhead but higher-latency recovery, as well as replication of key pointers, which reduces performance but enables fast recovery. Finally, Section 4.3 presents an experimental evaluation of the performance and accuracy properties of the resilient versions of the major routines used by our target applications, under a wide range of inputs and error injection rates.

4.1 Error Detection

4.1.1 Algorithmic Detection

Each algorithm used by the above application was enhanced with an algorithm-specific checker that validated its results by exploiting some algorithmic identity. Each checker computes some aggregate property of the algorithm's results using two different algorithms to produce vectors v and v'. If their relative difference by more than some threshold τ, the checker declares that an error has occurred. The relative difference is computed as $\frac{||v-v'||}{max(||v||,||v'||)}$. Each error detection causes the routine's execution to be restarted using the rollback mechanism described in Section 4.2.

Matrix-matrix multiplication (MMM).

The operation $A \cdot B$ is checked using a matrix vector multiplication (MVM), via the identity: $(A \cdot B) \cdot x = A \cdot (B \cdot x)$, where x is an error-checking vector. Since we set the check vector x to be all 1s this is equivalent to both sides computing the sums of the columns of matrix $A \cdot B$ using a single MVM on the left and two MVMs on the right. Most errors in the computation of $A \cdot B$ will affect the sums of its columns and will thus be detected. This check is efficient because the checker is asymptotically faster than MMM, with MVM taking $O(n^2)$ time and MMM $O(n^3)$ time.

Matrix-vector multiplication (MVM).

The MVM algorithm $Ax = b$ is checked using a similar identity $(x^T A)x = x^T(Ax) = x^T b$. We again set x to be the vector of 1's, making the product $x^T A$ equivalent to sum of the columns of matrix A. Although the computation of $x^T A$ involves additions rather than multiplications, the complexity of computing $x^T A$ is $O(n^2)$, the same as the original MVM operation. However, for applications such as ADDR where the MVM is applied to same matrix many times with different vectors, the column sums can be reused, amortizing the cost of this computation. Our experimental evaluation below thus focuses on the overhead of the $x^T(Ax)$ computation, which has $O(n)$ complexity.

Symmetric Rank-K update (SYRK).

The SYRK update is a special case of matrix-matrix multiplication $B = \alpha A \cdot A^T + \beta B$, where an $n \times k$ matrix A is multiplied by itself to produce an $n \times n$ matrix with rank k, which is incremented into the $n \times n$ matrix B. Its results are checked using the same algorithm as MMM by checking the identity $Bx = \alpha A \cdot (A^T x) + \beta Bx$, where x is the vector of all 1's. This check runs in time $O(n^2)$, as compared to the $O(n^3)$ cost of SYRK.

Cholesky Decomposition (CD).

In this decomposition matrix A is decomposed into $L \cdot L^T$ where L is a lower-triangular matrix with a positive diagonal. This routine is checked just like MMM by checking the identity $Ax = L \cdot (L^T x)$, which is cheaper than the $O(n^3)$ complexity of the deterministic CD algorithm. The GSL implements an iterative algorithm that runs faster than $O(n^3)$ time but as shown in the experiments below, it is significantly slower than our checker.

Fast Fourier Transform (FFT).

The FFT algorithm decomposes a function into a sum of sine waves of different frequencies: $f(x) = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$ for some constant k. We check the results of FFT by using Parseval's theorem, which states that $\sum_{n=0}^{N-1} |x[n]|^2 = \frac{1}{N}\sum_{k=0}^{N-1} |X[k]|^2$, where x is the original function and X is its transform. Intuitively it means that the energy of the original function is preserved by the transform. GSL implements optimized radix 2, 3, 4, 5, 6 and 7 subtransform routines as well as a general radix-N FFT routine. The subtransforms are optimized and consume $O(nlog(n))$ time and the general one consumes $O(n^2)$ time. Therefore, GSL has a scheduling algorithm that factorizes the length of the input FFT, running the subtransforms whenever possible, using the fallback general radix-N routine only for the remaining factors. The theorem applies to FFT of any radix and provides an $O(n)$ time check for the $O(nlog(n))$ or $O(n^2)$ FFT algorithms.

Finite Impulse Response Filter Generation (FIR).

The FIR filter generation algorithm used in DRC generates a series of samples over the function $sinc(x) = \frac{sin(x)}{x}$ and modulates it with a Blackman window. This function has the property that $\int_{-\infty}^{\infty} sinc(x)dx = 1$. The Blackman window - modulated output of this routine preserves this same property for most of the time, making it possible to check the output by computing the sum and comparing it to 1. Computing the sum requires $O(n)$ additions, as compared to the $O(n)$ trigonometric function evaluations of the non-iterative FIR generation algorithm.

Runge-Kutta PDE Solver (RK).

RK is a 4-th order Runge-Kutta method for solving Ordinary Differential Equations of the form $\frac{dy}{dx} = f(y, x)$. It advances the variable x by steps of size h and uses estimates of the derivative $\frac{dy}{dx}$ at each point x to find the value of y at the next point $x + h$. The implementation of RK in GSL uses adaptive step-size control where it simultaneously performs the RK method with two step sizes h and $\frac{h}{2}$ (more precise). If the difference between the results obtained with the two step sizes are larger than a threshold τ, the algorithm switches to the smaller step size to maintain accuracy and resumes. If it is smaller than $\frac{\tau}{2}$, the algorithm switches to a larger step size. Because soft errors will usually cause significant differences between the two computations, the adaptive step size control algorithm is naturally fault toler-

ant. It will react to such errors by rolling back, temporarily increasing its step size and then reverting to the original step size when it observes that this is safe. As such, we utilize this algorithm to protect the computations within individual RK steps from soft errors. However, computations in user-provided function derivative functions are not checked since their invariants are notknown.

4.1.2 Memory Fault Detection

Detecting errors in application data structures requires checkers dedicated for this purpose. Depending on the algorithm's properties different error detectors are appropriate. In our work we considered the use of access protection hardware and replication of key data.

Since modern systems use 64-bit addressing the range of values that can be assumed by a given pointer is much larger than the amount of memory that can ever be assigned to an application. This means that there is a very high probability that the corruption of a given pointer will cause it to point to an unallocated address. Once this address is accessed the resulting violation will be detected by the memory protection hardware and communicated to the application via a Segmentation Fault or Bus Error signal. This mechanism is cheap but suffers from a long delay between an error and its detection.

An alternative mechanism is to replicate key application variables and to compare the values of the replicas on each access to the variable. While more expensive, this approach is effective if the number of key variables is small and the cost of rolling back computation is high.

The low cost of hardware memory protection mechanisms means that they are deployed in protecting all our routines. In contrast, replication is only used to protect the RK routine because it is more sensitive to soft errors than the others and the low detection latency provided by replication ensures that any state corruptions are quickly corrected and have little impact on execution time. This is important for resilience as well as performance since slowdowns due to frequent rollbacks increase the total number of errors the application is exposed to.

4.2 Recovery via Checkpoint-Restart and Replication

When errors are detected we employ a recovery method based on checkpoint-restart, where routine state is recorded at key locations and whenever an algorithmic or memory error is detected we roll back to that point in its execution. Rollbacks are implemented via the `sigsetjmp/siglongjmp` [9] routines. `sigsetjmp` records the application's execution state (registers and stack pointer) at the time of the checkpoint. On the detection of an error `siglongjmp` returns the application to the same execution point, unrolling any function calls between the checkpoint and the error detection. This method is useful for rolling back to checkpoints recorded in the same function or in a caller function. When execution returns to the `sigsetjmp` call the checkpointed portions of the routine's state are recovered and execution repeats.

Checkpoint data is protected using two different mechanisms: Pointers and scalars are replicated such that the checkpoint contains two copies of each. Arrays of floating point numbers are protected via a block-checksum-based error correcting approach. By treating the input array as a matrix and comparing row sums and column sums of the submatrices of that matrix, it detects the existence of errors and fixes them. This algorithm takes an array of numbers as input, divides it into blocks of $N \times N$ numbers and for each block it computes and stores the row and column sums, which are used for recovery.

The structure of each routine determines how checkpointing is deployed. All routines except RK are monolithic code blocks that take a relatively large amount of data as input, operate on it and then return. In contrast, RK takes in very little input and operates as a long sequence of steps, each of which involves a call to a user-provided derivative routine. As such, we checkpoint the state of non-RK routines immediately after the call to the routine and include in this checkpoint the routine's arguments and contents of the floating point input buffers. The one exception is the FIR routine, which takes no input and thus requires only the processor's state to be checkpointed. Because RK's checkpoints are very small and it is more sensitive to errors, it is checkpointed periodically during its execution with a fixed number of steps between checkpoints. As our experiments in 4.3.2 show, the choice of the RK checkpoint period significantly affects its efficiency with longer periods suffering from more overhead due to rollbacks and short period spending more time checkpointing.

4.3 Evaluation of Subroutine Resilience

4.3.1 Linear Algebra and Integer Routines

This section evaluates the routines MMM, MVM, SYRK, CD, FFT and FIR.

Overheads.

We evaluate the costs of the above algorithmic error checkers by measuring their costs and effectiveness when running at different error rates on different types of inputs. Figure 1 measures the overheads of employing the algorithmic error detectors when no errors are injected, reporting the average of 100 runs. The linear algebra routines were executed on $n \times n$ square matrices where n varies from 100 to 1000. FFT ran on inputs vectors with 128K to 16M entries and FIR on window widths of 1M to 8M. RK ran on an integral with intervals $[0, t_1]$ where $t_1 \in \{10, 15, 20, 25, 30, 50, 75, 100\}$. Its ODE system is a second-order nonlinear Van der Pol oscillator equation. Our experiments show that the cost of algorithmic error detection is low ($< 10\%$) for small and drops with increasing input sizes.

Figure 2 shows in the bars the overhead of all resilience techniques with no fault injection, separating into overheads due to algorithmic error detection, the checkpointing of routine inputs as well as checkpointing processor state. The data shows that checkpointing the input of routines consumes more time than algorithmic error detection and that the cost of checkpointing processor state is negligible. In all cases, these overheads decrease as the input set size increases.

Output Accuracy.

We now evaluate the effectiveness of our algorithmic detectors when errors are injected. In each case when an error is detected by our algorithmic detectors the routine's execution would restart from the beginning upto five restarts. Figures 3 show the fraction of entries in the outputs of each

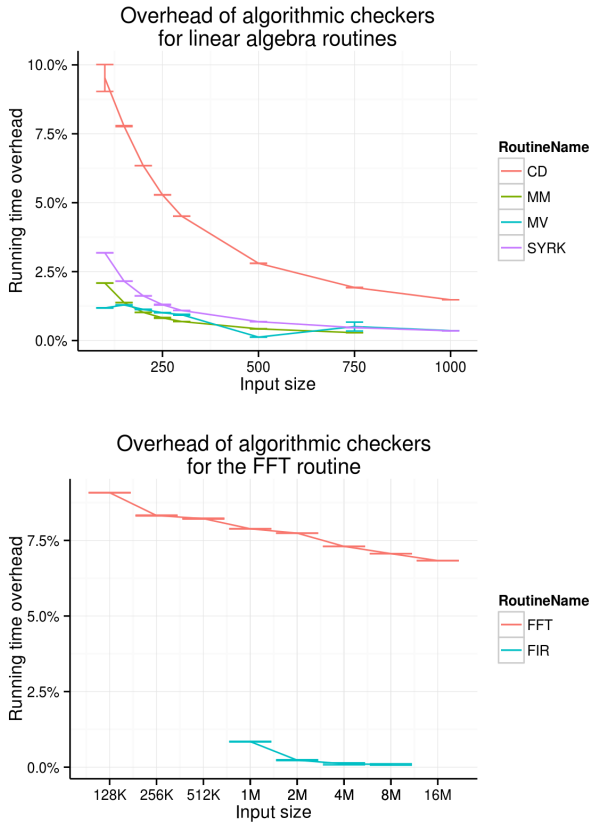


Figure 1: Overhead of resilience techniques when errors are not injected

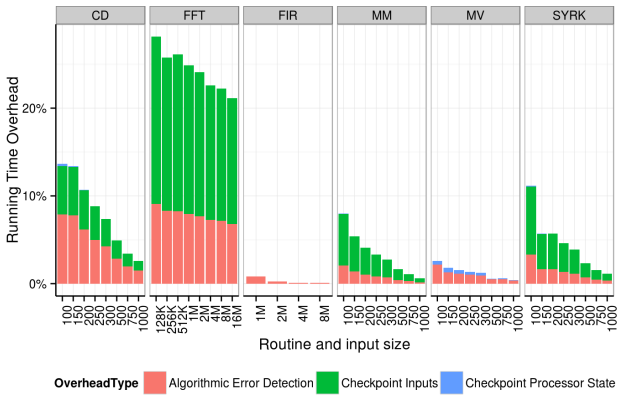


Figure 2: Overhead of resilience techniques. Results when no errors are injected are shown.

routine (y-axis) that have an error of a given magnitude (x-axis) as error magnitudes span from 1e-14 to 1e-4. This figure is a focused view on few representative cases (each routine run on a single input size and error injection rate) across different resilience configurations: no error detection as well as three error detection thresholds. As the error magnitude increases the fraction of output entries with er-

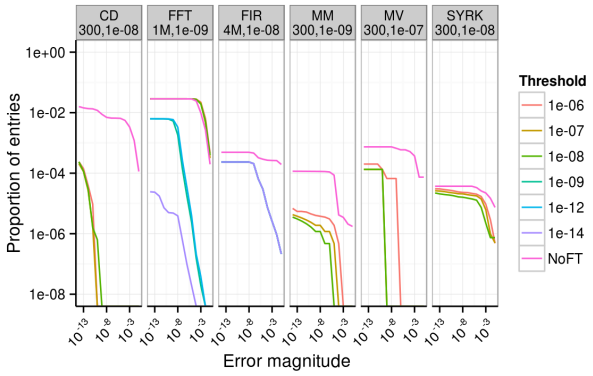
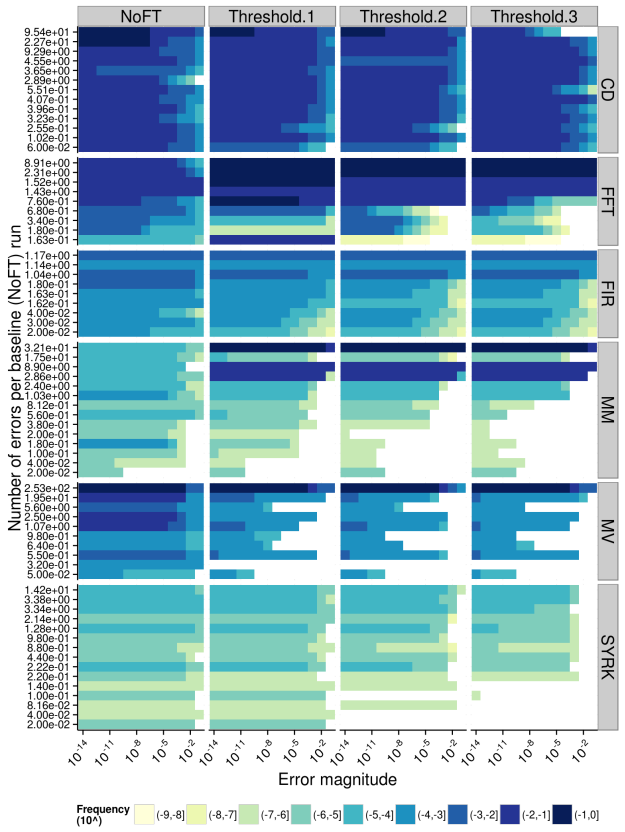


Figure 3: Error magnitude histogram showing the effectiveness of the error detector



Routine	Threshold 1	Threshold 2	Threshold 3
CD	1e-06	1e-07	1e-08
FFT	1e-06	1e-09	1e-14
FIR	1e-06	1e-09	1e-14
MM	1e-06	1e-07	1e-08
MV	1e-06	1e-07	1e-08
SYRK	1e-06	1e-07	1e-08

Figure 4: Error magnitude histogram of the entire configuration space

rors of that magnitude drops. Importantly, the fraction of erroneous entries is reduced at all magnitudes as a result of error detection and restart. Further, this reduction is more significant as the detection threshold shrinks since smaller magnitude errors are detected.

Figure 4 shows the same information but for all experimental configurations: (i) input sizes as above (ii) error injection rates of 1e-7 to 1e-10. The graph is a grid, with results for different routines listed vertically and different resilience configurations (no detection and the different detection thresholds shown in the table at the bottom) listed horizontally. The graph for each routine and resilience configuration presents the curves from Figure 3 as a heatmap. The y-axis shows different routine input sizes and error injection rates, sorted according to the average number of error injections per non-fault injected run, in decreasing order from top to bottom. The x-axis shows the different error magnitudes and the shade of each point is the fraction of the routine's entries that have a given error magnitude, with darker shades indicating more such entries (each shade corresponds to a fraction that is some power of 10). Thus, each row of Figure 4 represents the same information as each curve in Figure 3, with the shades of the squares representing the same information as the height of the line's points.

The data shows that in general as the error detection threshold drops there are fewer entries that have an error of a given magnitude. However, there are exceptions where the fraction of erroneous entries rises as the detection threshold shrinks (e.g. MMM and FT between NoFT and Threshold 1). This is caused by the fact that we restart each routine on an error detection, which can causes the routine's execution to be exposed to additional error injections. As the threshold is further tightened the amount of error in the output shrinks, indicating that error detection is ultimately effective if it is configured to be sufficiently accurate. Another phenomenon that is observed is that as the number of errors injected in a given run decreases the degree of error in routine outputs also generally decreases. However, this is not a fully consistent phenomenon, with some configurations that have more error injections being less vulnerable than others with a higher number of injections because the former had a higher injection rate but the latter a larger input size. This indicates that while the number of injected errors is strongly correlated with overall application vulnerability to errors, other properties such as input size also play a role.

4.3.2 *Runge-Kutta*

The difference between RK routine and others motivates special discussion of its resilience properties. Since RK's adaptive step size mechanism is already effective at tolerating many numerical errors we used it as the error detector for this routine without adjusting its tolerance parameters. The configurable portions of the RK resilience method are thus the choice of checkpointing and/or replication strategy. Configurations `Ckpt` and `NoChkpt` denote whether checkpointing was used and configurations `Re` and `NoRe` denote whether replication was used. We consider checkpointing periods equal to 1, 10^2, 10^3, 10^4 and 10^5 calls to the RK stepping function. Further, we consider the following replication configurations: (i) "No replication", (ii) 'Medium replication", where routine pointers are replicated but the replicas of each pointer are compared at a single code location where it is used, and (iii) "Maximum replication", where the comparison is performed on every pointer use.

Overheads.

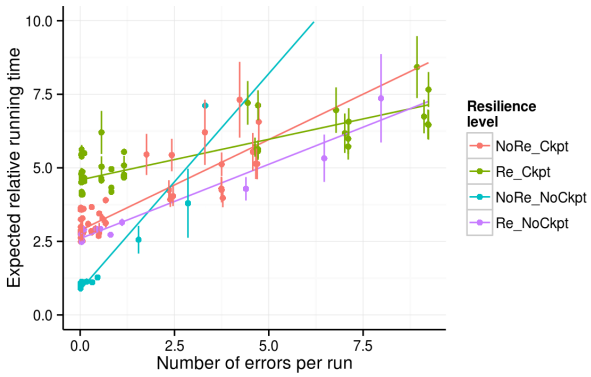


Figure 5: Overhead of resilience techniques for RK when errors are injected

Figure 5 shows the RK's overhead when between 0 and 10 errors per run are injected.Although the `NoRe_NoCkpt` configuration has the lowest overhead when only a few errors are injected (low error rate or small execution times), as the number of errors per run increases this configuration quickly becomes extremely slow because an error is detected in almost every run. Since each such detection requires a restart of the routine, at higher error rates this causes RK to restart many times before it completes. In contrast, while the other configurations have a higher overhead for a few errors per run, their execution time scales much better with increasing error rates, making them significantly more practical. In particular, the overheads of the configurations with more replication and checkpointing scale better as the number of errors per run increase. By varying the checkpointing interval and the degree to which various RK pointers are replicated it is possible to achieve a wide range of design points that provide the best performance at various error rates.

Output Accuracy.

Figure 6 shows the fraction of errors in RK's outputs for various checkpointing and replication configurations, in the same format and using the same experimental setup as in Figure 4. It shows that while the use replication reduces the amount of error in RK's outputs, increased checkpointing doesn't have a significant effect on this accuracy metric. Figure 7 evaluates the effect of checkpointing on accuracy from another perspective: the fraction of RK runs where the root-mean-squared error of the output vector is below 1e-7 (y-axis), relative to the number of error injections per run (x-axis). It shows that RK configurations with higher level of resilience and in particular, more frequent checkpoints, are able to achieve this accuracy target more frequently. This is primarily because by avoiding frequent rollbacks due to crashes these techniques reduce the total number of errors that are injected into the application run, which also reduces the probability that one such error will be missed by RK's algorithmic error detector. As with overhead, the choice of checkpointing interval and replication degree make it possi-

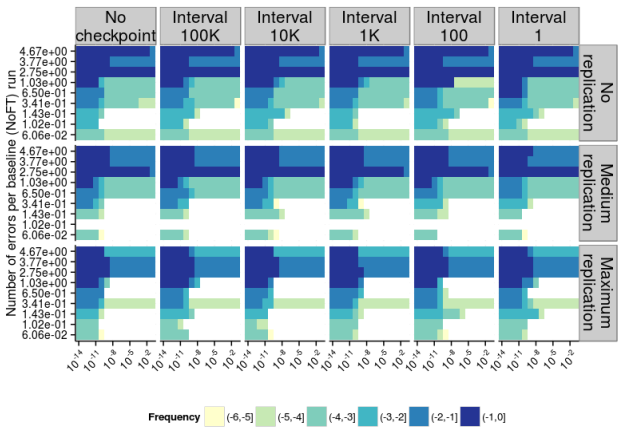


Figure 6: Error magnitude histogram of the entire configuration space of RK

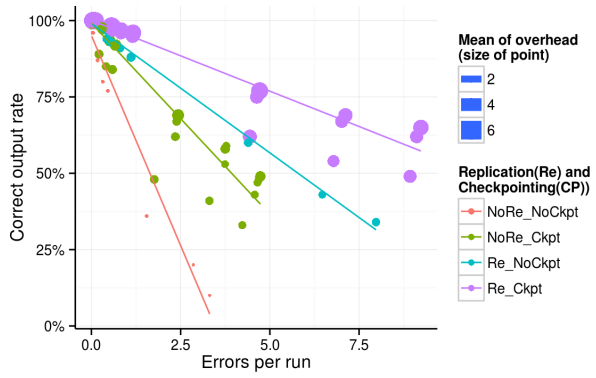


Figure 7: Fraction of RK runs that achieve low error (smaller than 1e-7).

ble to choose a wide range accuracy levels and probabilities that these targets will be satisfied.

5. EVALUATION OF RESILIENCE METHODOLOGY IN APPLICATIONS

Having evaluated the resilience properties of the individual application routines and resilience techniques we now consider our three target applications ADDR, Hattrick and DRC. Section 5.1 reports on the impact of errors and resilience techniques on application performance. Section 5.2 describes the distribution of errors in the results of these applications. These results are based on a fault injection campaign that includes a range of fault injection rates, input sizes and configurations of resilience mechanisms, with 100 application runs for each combination of parameters. This corresponds to a total of 66,100 fault injection experiments for ADDR (4-process parallel), 43,200 for DRC (sequential), 102,000 for Hattrick (sequential). Our experiments were performed on compute nodes with two 2.33 GHz dual-core Xeon E5345 processors with 4GB Ram and RHEL 4 OS.

A given application run may terminate successfully or crash due to an unrecoverable error. Unrecoverable errors occur when there is a segmentation fault in code outside our protected routines or if five rollbacks of a routine have been observed in a given run since such repeated rollbacks almost always indicate that the corruption in application state cannot be repaired. Since we assume that the user needs application results regardless of the error rate, if the application crashes it is restarted from the beginning as many times as needed for it to complete.

To evaluate the behavior of these applications in a broad range of error environments we consider several error injection rates. For reference, assuming a typical FIT of 100-100,000 for individual processing cores (FIT = failure in one billion hours of operation and this range has been experimentally observed for commodity electronics [16, 23]), an HPC system with one million cores will experience .1-100 errors per hour, which is equivalent to 3e-11 - 3e-14 errors per processor cycle on a 1Ghz processor. The range of error injection rates used used in our experiments were chosen such that (i) on average one or more errors were injected in the executions of our target applications and (ii) at least some application runs completed without crashing. Results for error rates that induce fewer than one error per run were extrapolated from our experiments with higher error rates. This was done by considering only the runs where exactly one error was injected and trivially scaling the results to correspond to 1 in every n runs being injected with an error while remaining $n-1$ runs complete with no injection.

5.1 Performance Impact

Our fault injection experiments ran each application configuration to completion, recording whether it completed and if so, the magnitude of the error. We calculate the execution time of each configuration, accounting for restarting the application in cases of unrecoverable failures via the formula $\frac{t_c}{1-p_f}$, where t_c is the average completion time of such runs and p_f is the probability of an unrecoverable failure. Figures 8, 9 and 10 show the expected execution time, relative to the execution time of the non-fault tolerant version of the application when no errors are injected. Each figure shows a separate graph for each input size and along with the ID of the input shows the application's execution time on that input to quantify its vulnerability to errors on the input. Data for the non-fault tolerant version is shown in red and the fault tolerant version with the optimal configuration for each input size is shown in blue. For ADDR we considered error checker thresholds from 1e-4 to 1e-7. For DRC we considered thresholds from 1e-6 to 3e-8. Finally with Hattrick we considered the use of one pair of copies for each pointer occurrence or using a different pair of replicas for each pointer occurrence, as well as checkpoint periods of 1, 1e3 and 1e5. The graphs from left to right correspond to different application input sizes. Within each graph the x-axis corresponds to error injection rates (the real-system region of 3e-11 to 3e-14 highlighted in grey) using a logarithmic scale, while the y-axis shows the relative execution time using a linear scale.

Figure 8 shows that the non-fault tolerant ADDR slows down from 2x to 6x, depending on the input set size, as a result of the injected faults. The optimal fault tolerance version the algorithm shows degradation that are around one half of the non-fault tolerant one, which implies that our technique improve ADDR's resilience by 50%. However,

for lower error rates the difference between the two versions is small as both perform well.

Figure 9 shows that since DRC performs many more integer operations and memory copies than ADDR, it is more vulnerable to errors that corrupt the values of pointers or cause loads and stores to target invalid memory locations. Such errors cause memory segmentation faults, which force DRC to restart its execution and thus incur a very high overhead. That is the reason why the performance slowdown of its non-fault tolerant version is much larger than for ADDR at high error rates. In contrast, the optimal fault tolerant version almost completely eliminates the performance impact of faults. However, at low error rates the cost of resilience is sufficiently high that at low error rates the non-fault tolerant version of DRC performs slightly better.

Figure 10 shows that the significant cost of pointer replication causes the fault tolerant version of Hattrick to be consistently slower than the non-fault tolerant one in most cases. As shown below, the main benefit of resilience mechanisms is to improve the accuracy of Hattrick's results.

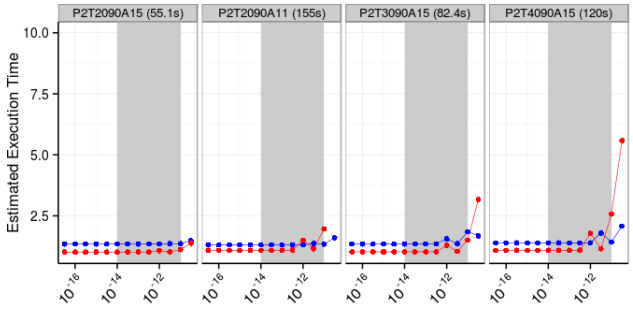


Figure 10: Execution time of Hattrick relative to non-fault tolerant fault-free execution time. Results for the non-fault tolerant version (red) and the optimal fault tolerant version (blue) are shown.

5.2 Result Accuracy

If the probability that a given run completes without an unrecoverable error is greater than 0 then the application will eventually complete with some output after some number of restarts. This section evaluates the distribution of errors in the resulting outputs. The output of ADDR is the vector that holds the solution to the linear system. For Hattrick it is the vector that holds the state of the n-body system at the final application time-step, with 6 numbers per body Finally, DRC's output is a Pulse Code Modulation-formatted file, which is a sequence of numbers. Since the outputs of each routine can be represented as a vector we calculate the error in the outputs by comparing the vector v_e returned by each application when errors are injected to the correct vector v_c computed during a fault-free run and computing the L2 norm of the difference: $\sqrt{\frac{\sum_i (v_c - v_e)^2}{|v_c|}}$.

Figures 11, 12 and 13 show the errors in ADDR, DRC and Hattrick outputs from a user-centric perspective. They plot the fraction of application runs for which a numerical error equal or smaller than 10^{-8} is achieved. From left to right each figure shows graphs for different input sizes and from top to bottom it shows results for different error magnitude bounds. The x-axis in each graph shows the fault injection rate while the y-axis shows the fraction of runs where the error target was satisfied. Both axes are expressed in logarithmic scale. Each graph shows data for the non-fault tolerant version of the application (all points connected by the line) as well as all the fault tolerant versions (optimal configurations connected by a line).

The data shows that the probability that the non-fault tolerant version of each one the three application violates the user-provided error bound is above 10% for Hattrick and DRC in the region of realistic error rates (3e-11 to 3e-14) and above 1% for ADDR across all inputs. When resilience is applied to these applications the probability of violating the error bound drops to below 1% for ADDR and Hattrick and below 10% for DRC in the same region. This shows that the use of all the resilience techniques is critical for ensuring that application outputs are valid with a high probability even on very large HPC systems.

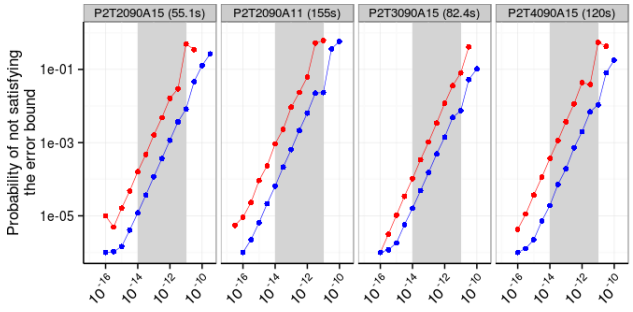


Figure 13: Probability of not satisfying the user-provided error bound for Hattrick. Results for the non-fault tolerant version (red) and the optimal fault tolerant version (blue) are shown.

6. SUMMARY

In this paper we show that it is possible to effectively protect three real scientific applications from a wide range of faults by properly combining three different fault tolerant mechanisms: Algorithmic error checks, replication of critical data structures and checkpoint-restart of individual routines. We demonstrate that very significant improvements can be achieved in terms of reduction of performance slowdown and output numerical accuracy when scientific problems face error injection rates from $3 \cdot 10^{-11}$ and $3 \cdot 10^{-14}$ per processor cycle at 1Ghz, which are the expected rates in future high performance computing systems. This work demonstrates that current scientific application can be safely executed on such systems if the programmers comprehensively combine a range of resilience techniques.

7. REFERENCES

[1] R. C. Baumann. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, September 2005.

[2] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed Optimization and Statistical Learning via the Alternating Direction Method of

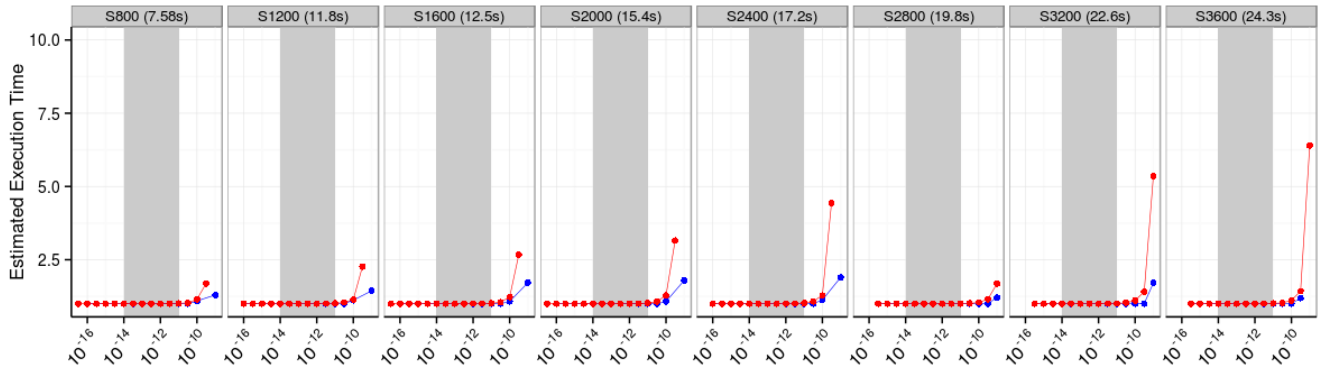


Figure 8: Execution time of ADDR relative to non-fault tolerant fault-free execution time. Results for the non-fault tolerant version (red) and the optimal fault tolerant version (blue) are shown.

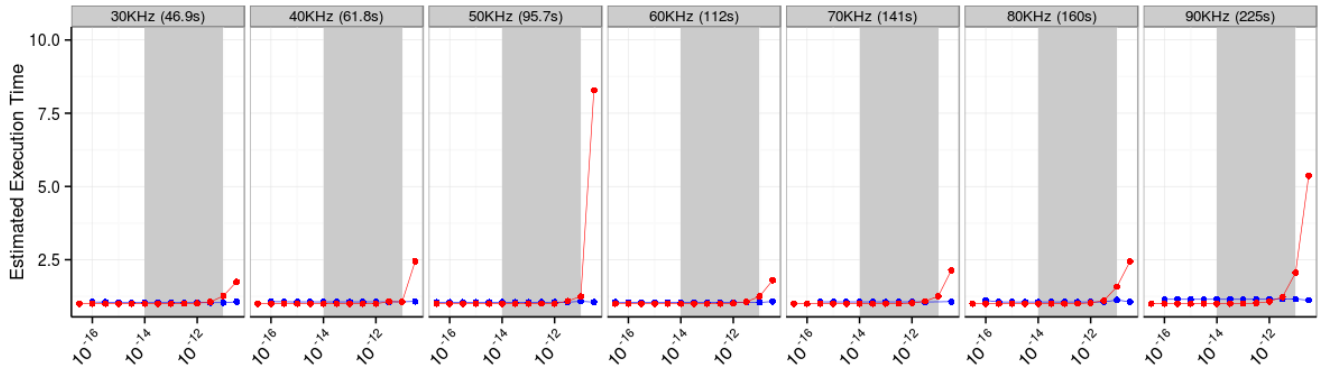


Figure 9: Execution time of DRC relative to non-fault tolerant fault-free execution time. Results for the non-fault tolerant version (red) and the optimal fault tolerant version (blue) are shown.

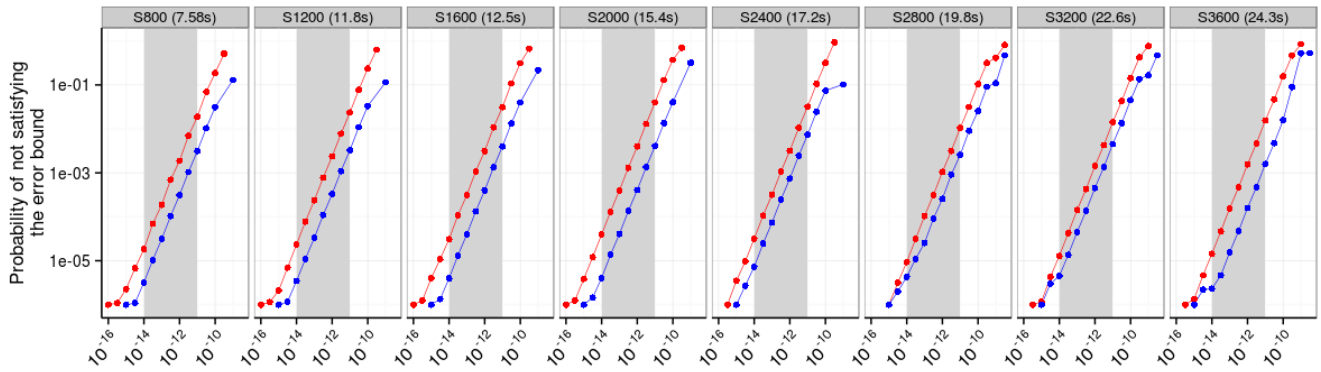


Figure 11: Probability of not satisfying the user-provided error bound for ADDR. Results for the non-fault tolerant version (red) and the optimal fault tolerant version (blue) are shown.

Multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, June 2011.

[3] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz. Fault Resilience of the Algebraic Multi-Grid Solver. In *International Conference on Supercomputing*, pages 91–100, 2012.

[4] C. da Lu and D. A. Reed. Assessing Fault Sensitivity in MPI Applications. In *Supercomputing*, November

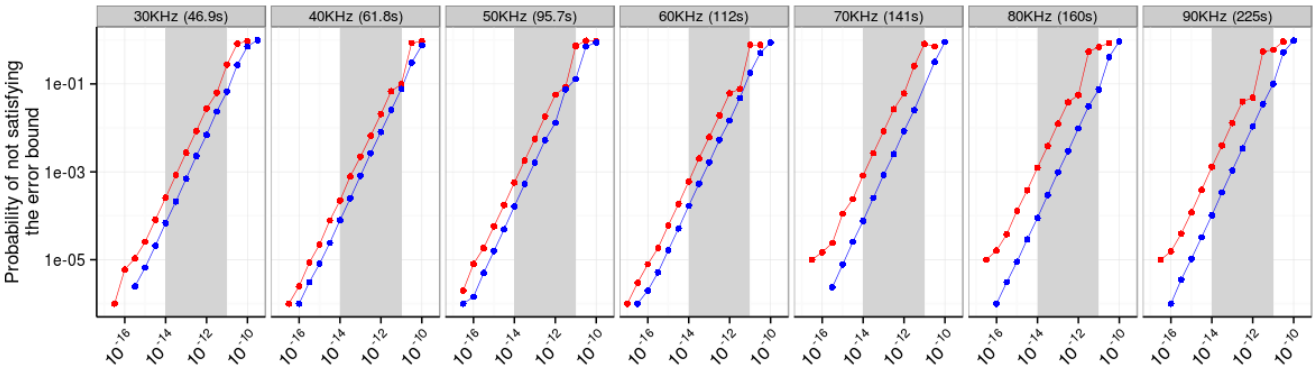


Figure 12: Probability of not satisfying the user-provided error bound for DRC. Results for the non-fault tolerant version (red) and the optimal fault tolerant version (blue) are shown.

2004.

[5] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *International Symposium on Computer Architecture (ISCA)*, 2010.

[6] V. Degalahal, N. Vijaykrishnan, M. Irwin, S. Cetiner, F. Alim, and K. Unlu. SESEE: A soft error simulation and estimation engine. In *MAPLD International Conference*, 2004.

[7] K. Ferreira, J. Stearley, I. James H. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Supercomputing*, 2011.

[8] F. S. Foundation. Gnu scientific library – reference manual, 2011.

[9] T. O. Group. The single unix (r) specification, version 2, 1997.

[10] M. C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, November 1997.

[11] K.-H. Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. In *International Conference on Dependable Systems and Networks (DSN)*, pages 161–170, 2010.

[12] F. Irom and F. F. Farmanesh. Frequency dependence of single-event upset in advanced commercial powerpc microprocessors. *IEEE Transactions on Nuclear Science*, 51(6):3505–3509, December 2004.

[13] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *International Conference on Dependable Systems and Networks (DSN)*, pages 317–326, 2007.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[15] H. Li, J. Mundy, W. Patterson, D. Kazazis, A. Zaslavsky, and R. I. Bahar. Thermally-induced Soft Errors in Nanoscale CMOS Circuits. In *IEEE International Symposium on Nanoscale Architectures (NANOARCH)*, pages 62–69, 2007.

[16] X. Li, M. C. Huang, K. Shen, and L. Chu. A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility. In *USENIX Annual Technical Conference*, 2010.

[17] L. W. Massengill, B. L. Bhuva, W. T. Holman, M. L. Alles, and T. D. Loveless. Technology Scaling and Soft Error Reliability. In *IEEE Reliability Physics Symposium (IRPS)*, pages 3C.1.1–3C.1.7, 2012.

[18] S. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3), September 2005.

[19] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *International Symposium on Microarchitecture (MICRO)*, December 2003.

[20] W. T. Olson. Hattrick n-body simulator. http://code.google.com/p/hattrick-nbody, 2012.

[21] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *International Symposium on Computer Architecture (ISCA)*, pages 25–36, 2000.

[22] D. Sbragion. Drc: Digital room correction. http://drc-fir.sourceforge.net/, 2012.

[23] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *SIGMETRICS*, 2009.

[24] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A Numerical Optimization-based Methodology for Application Robustification: Transforming Applications for Error Tolerance. In *International Conference on Dependable Systems and Networks (DSN)*, pages 161–170, 2010.

[25] J. F. Ziegler. Terrestrial cosmic rays. *IBM J. Res. Dev.*, 40(1):19–39, Jan. 1996.

Prepared by LLNL under Contract DE-AC52-07NA27344.