The Journal of Supercomputing (2022) 78:18691–18724
https://doi.org/10.1007/s11227-022-04625-x

Studying error propagation on application data structure and hardware

Zuhal Ozturk[1] · Haluk Rahmi Topcuoglu[1] · Mahmut Taylan Kandemir[2]

Accepted: 22 May 2022 / Published online: 13 June 2022

Abstract

As technology scales, transistors become smaller and aggressive power optimization techniques combined with high operation frequencies and performance-enhancing microarchitectural techniques are employed to achieve increasingly higher performance and power efficiencies. Unfortunately, these developments make the modern systems more vulnerable to soft errors, which are becoming a critical issues in both hardware and software domains. Motivated by this observation, in this work, we propose, implement, and evaluate two error propagation metrics in order to characterize error propagation at both software and hardware levels. The first metric aims to measure error propagation on program data structures, whereas the second one measures the fraction of corrupted locations in the cache memory structure for a given period of time. We evaluate our proposed metrics by performing an empirical study of two application programs using both single-threaded and multi-threaded executions, and varying various experimental parameters such as thread count, error rate, location of errors, and architectural parameters. Our extensive experimental analysis reveals that error propagation over program data structures is highly dependent on application behavior.Further, depending on the cache parameters used, propagation of errors on cache can exhibit different patterns. This paper also discusses how our observed error propagation trends in program data structures and data caches are correlated with each other, focusing in particular on the differences in error propagation speeds in application data structures and data caches.

Keywords Soft errors · Error propagation · Fault injection · Reliability · Multi-threading

✉ Haluk Rahmi Topcuoglu
haluk@marmara.edu.tr

Extended author information available on the last page of the article

1 Introduction

As process technology scales, transistors become smaller, and this makes it possible to squeeze more compute and storage components into the same chip area. As a result, state-of-the-art processors include multiple CPU cores, integrated GPUs, multi-layer cache hierarchies, on-chip networks, and multiple on-chip memory controllers, operated at high frequencies. Such smaller and denser components, higher operation frequencies, and aggressive power and performance optimization techniques in turn make such processors increasingly more vulnerable against soft errors, as discussed in the literature [1–6].

Soft errors are transient events that manifest themselves as bit flips in memory or logic circuit outputs, and can cause application program and/or system software to move to an undesirable state (e.g., operating system can crash, application program can hang, or worse, can return wrong results). Cosmic radiation, alpha particles from packaging materials, voltage fluctuations, and high temperatures are among the chief reasons behind this type of errors [7–13]. Since the emergence of first works on soft errors, architects and software designers started to explore trade-offs between performance and reliability, as many known measures against soft errors lead to various performance implications [14].

As soft errors become increasingly important, various fault tolerance techniques have been developed and tested, with the goal of countering the impact of these errors on program execution. These fault tolerance techniques are implemented at hardware and/or software levels, and they introduce extra cost and/or performance overheads. While the hardware-level fault tolerance techniques add extra components to the system to improve reliability [15–19], the software-level mechanisms are mostly based on time and information redundancy [20–24]. Additionally, several so called integrated or cross-layer techniques that consider reliability problems in both hardware and software layers have also been proposed by prior art [25–29].

Although these techniques can be very effective in many execution scenarios and environments, most of them have a critical drawback – they employ Soft Error Rate (SER [30]), as their primary optimization metric. Soft Error Rate represents the error rate of the components due to unrecoverable errors. SER is an important metric; however, it is a summary/average metric and does not give any information about the dynamic behavior of the errors. Consequently, SER is difficult to employ in a scenario where one wants to provide "just enough reliability" with minimum cost. We believe that understanding the *error propagation behavior* of an application program running on a given hardware is key for judicious employment of available error propagation techniques, instead of relying on summary-based metrics such as SER and its variants [30–34].

We want to emphasize that error propagation can be studied at *both hardware and software levels*. Firstly, it is important to know how much of a given hardware component is corrupted (i.e., contains erroneous data) at any given point in execution. For example, in the context of data caches, by knowing the fraction of cache locations that are corrupted at a particular point in execution, hardware

fault tolerance techniques may be utilized more efficiently (e.g., by tuning their aggressiveness). Secondly, it is also important to understand which parts of application data[1] are more prone to errors, and how errors propagate over application data structures during the course of execution. Thirdly, it is important to *correlate* error propagation in program data structures with error propagation in hardware. Finally, it can be interesting to explore what new optimization strategies can be developed once the error propagation information is exposed to compiler.

Soft errors may result in hang, crash, or silent data corruption (SDC). Among these possible outcomes, SDC is arguably the most harmful case since in SDC the program does not give any sign of error [35], and it can go undetected for a long time. Clearly, since SDC leads to a waste of time and resources, especially for the programs with long execution times, it is a very critical issue and thus our study is limited to errors resulting in SDC. In our study, error propagation is discussed in terms of application data. Therefore, errors that will terminate the program are not injected to the system. We only inject single and double bit errors to the system that cause SDC and observe their propagation across data structures and cache locations. That is, unlike previous works, our work studies error propagation across *both* program data structures and hardware (cache). Also, since the errors can manifest themselves after a long execution time, we utilize an accelerated error injection rate to observe the impact of errors within the execution time of our applications. Similar accelerated error injection has been used in prior studies as well [36–39].

The main contributions of our paper can be summarized as follows:

- We propose an error propagation metric that evaluates the propagation of soft errors at a program data structure level. This metric captures both the speed at which the error propagates within and across data structures as well as the amount of error accumulated up to a point in execution. We study this metric, for both single-threaded and multi-threaded applications, by changing various parameters such as error rate and set/amount of data elements where errors initially occur. Our experimental evaluations with two application programs, Sparse Matrix Vector Multiplication (SpMV) and Preconditioned Conjugate Gradient (PCG), indicate that the applications and their inputs affect the error propagation. In the SpMV application, the error is propagated progressively across different data elements, but its pattern strongly depends on the input given to the application. Although the error propagates up to 70% of the result vector for several matrices, limited propagation is observed for others (up to 20% of the result vector). The effect of different inputs on the propagation curves can also be seen from the PCG application. For several matrices, the error propagates to limited part of the result (between 20 and 40%), while in others the entire result is calculated incorrectly.
- We propose an error propagation metric that evaluates the propagation at a data cache level. For that, we perform fault injection experiments with different appli-

[1] Clearly, OS data structures can also be corrupted; but, in this work, we exclusively focus on application data structures.

cations. In addition to the error propagation characteristics, we also study the cost of error at the hardware. Specifically, we measure the error free data loss due to erroneous data in cache memories by conducting an experimental study. Compared to the error propagation across data elements and structures, the error propagation on data caches exhibits quite different patterns. Because of the cache replacements, not all the erroneous data are present in the cache at all times, therefore fluctuations can be observed the error propagation curves.

- We discuss the correlation between error propagation on application data structures and error propagation on data caches. Our results also reveal a high correlation between error propagation on program data structures and error propagation on data caches. Specifically, the correlation coefficient for certain inputs of the SpMV application is 0.9 or above (for matrices including bcsstk07, nos5, msc00726, and plbuckle). While the calculated correlation coefficients are lower when the applications' data structures cannot fit in the caches, error propagation in the software and hardware layers follows a similar trend when the program data fits in the caches more easily.

To our knowledge, this is the first detailed study of soft error propagation across data elements and data cache, via specially-defined error propagation metrics, for both single-threaded and multi-threaded executions. The results from this work can be used by both software designers (e.g., to decide which data structures to protect) and hardware designers (e.g., which cache locations to harden).

The rest of the paper is organized as follows. Section 2 summarizes previous studies. In Section 3, we define the error propagation metrics that we propose for hardware and software level propagation and also discuss the details of our fault injection framework. Section 4 presents the details of experimental study and Sect. 5 presents the results of the empirical study for both our proposed metrics. Finally, Sect. 6 summarizes the paper and discusses possible future work.

2 Related work

We classify the related work of this study in three categories – vulnerability factors, error related metrics, and error propagation analysis. Section 2.1 summarizes the proposed vulnerability metrics, and Sect. 2.2 summarizes proposed fault propagation metrics. Section 2.3 discusses Silent Data Corruption analyses for different applications.

2.1 Vulnerability factors

In the literature different vulnerability factors are proposed to evaluate the error vulnerability at different layers [40–48]. In general, vulnerability factors describe the probability with which an internal error affects the output [40]. Architectural Vulnerability Factor (AVF) represents the probability with which a fault in the processor structure would have an effect on the output [40, 41]. The vulnerability of

registers and caches cannot be computed using AVF alone, due to their special characteristics. Therefore, Cache Vulnerability Factor (CVF) [42] and Register Vulnerability Factor (RVF) [43] have been proposed to evaluate the vulnerabilities of these components. A vulnerability factor for ECC-protected memory is also proposed [44]. The False Error Aware (FEA) vulnerability metric measures masked errors among *detected* and *uncorrected errors* in memory. The Program Vulnerability Factor (PVF) is proposed to quantify the program vulnerability independent from the target architecture [45]. The vulnerability can be calculated for the resources that are visible at the software level with the PVF. Similarly, the Instruction Vulnerability Factor (IVF) performs vulnerability analysis in the software level [46]. However, instead of the architectural resources, the IVF estimates the output corruption due to an error in an instruction. Data Vulnerability Factor (DVF) evaluates the vulnerability of data structures and allows a fine-grained analysis [47]. Finally, Thread Vulnerability Factor is proposed to measure the vulnerability of a thread by considering data dependencies for multi-thread applications (TVF) [48].

While these previously-proposed metrics focus on the error transaction on the architecture (program/thread/instruction), our work examines how it spreads among the application data. Thus, information on how and where the error is propagated throughout the runtime is obtained.

2.2 Error related metrics

Several metrics associated with error propagation is proposed in the literature, including [49–52]. The *Permeability Metric* measures the error propagation in a modular system [49]. The study utilizes a black-box approach in which a system is assumed to consist of different modules that communicate with each other via their inputs and outputs. The error permeability is the probability with which an error propagates from a module's input to its output. In another work, the *Importance Metric* has been proposed with the goal of capturing the criticality of a given variable [50]. It is based on two sub-metrics – spatial impact metric, which measures the number of corrupted elements, and the temporal impact metric, which measures the amount of time during which the program state remains corrupted. The *Error Propagation Speed* (EPS) is defined as the number of processes per iteration that have some corrupted data [51]. The error propagation is modeled by utilizing a weighted graph, where a weight represents the probability with which the error propagates from the source node to the target node. The model is validated via fault injection experiments. Error propagation speed is also studied in another work. The *Speed Metric* measures the number of contaminated memory locations per time, and the *Depth Metric* measures the number of Message Passing Interface (MPI) processes that have contaminated data [52]. Beside error propagation metrics, error masking metric is also proposed [53]. In their study, error propagation is examined and error masking events are tracked (their proposed metric is equal to the frequency of occurrence of masking events).

Compared to these prior works, our proposed approach monitors the propagation of the error in application data structure space. In other words, the data structures

across which an error spreads are tracked. Thus, errors that affect the critical data can be analyzed individually. In addition, our proposed metrics are evaluated in *both* single-threaded and multi-threaded application programs.

2.3 Error propagation analysis

The error propagation is studied for different applications in the literature. The impact of the soft error on Partial Differential Equation (PDE) is another application-specific study [54]. Several PDE-based application and their underlying operation, sparse matrix-vector multiplication, are examined in detail, and the fault injection experiments show the error propagation pattern is related to the sparsity of the used matrix. In another study, the error resiliency of the GMRES iterative solver is studied [55]. Fault injection experiments are performed with three different GMRES implementations, and the impact of errors on the outputs are observed. The impact of the error is divided into four groups (crash, hang, successful, SDC), and implementations are compared in terms of error resiliency.

The error resiliency of HPC applications has been studied in the literature. For example, FlipTracker framework has been designed to analytically track error propagation and extract the patterns that provide natural resilience [56]. The framework models the application as a chain of code regions. Then, it examines the resiliency of the code regions by using data dependency analysis. It defines six computation patterns associated with resiliency: dead corrupted locations, repeated additions, conditional statements, shifting, data truncation, and data overwriting. These patterns tend to mask errors and increase the resiliency of the HPC applications.

Error propagation on the applications that employ the message-passing programming model has been investigated by several studies. Specifically, a framework has been developed to track Silent Data Corruption (SDC), and the error propagation in that framework has been defined as the spread of corrupted data from node to node via MPI communication [57]. Fault injection experiments have been performed on several applications exhibit different patterns – the error propagates progressively in several applications, but propagation is accelerated in certain periods in others. Moreover, for some applications, the corrupted data are not used or transmitted, so the corruption remains limited with the process that is directly injected with an error. In another study, the error propagation on various MPI-based applications is analyzed [35]. The error propagation and the impact of several parameters such as the data type, bit-flip location, number of bit flip on the error propagation are measured with fault injection experiments. Chaser is another framework proposed to study SDC propagation in MPI-based applications [58]. It is basically a fault injection tool that monitors soft error propagation. Note however, unlike our study, error propagation in Chaser is considered in terms of only faulty memory locations and read/write operations. In addition to soft errors, the error injecting method is used to understand error code propagation in MPI libraries [59]. In the study, the errors triggered error code bugs (unsaved error codes) are injected to MPI-based applications and runtime behaviors are observed.

Error propagation in memory components has also been discussed in the literature. The error propagation originated from the cache or processor is observed, and it is shown that 50% of the errors are masked without an error recovery mechanism [60]. In another work, error propagation originated from hardware components has been studied; but, unlike our study, it focuses on hard errors [61]. Their work uses lower-cost software-level fault tolerance methods to provide error resilience. The detection mechanism monitors the abnormal operating system (OS) and application behavior that can occur because of hard errors. It also provides error correction with software level methods such as check-pointing or replay.

The Support Vector Regression is another approach employed for error detection [62]. An SDC error detection strategy based on the support vector regression model is proposed, considering the instruction features determined from fault injection experiments. The proposed model is in turn used to guide the selective redundancy to improve the reliability. Since SDC is one of the most critical issues caused by soft errors, researchers are studied its impact by using fault injecting techniques. However, fault injection is time-consuming process, so different methods are proposed to accelerate the fault injection [63]. All fault injection sites in an application are analyzed systematically, and fault injection is performed for more vulnerable part of the application.

Besides the fault injection experiments, the propagation is also tracked through analytical models. For example, SDC probability has been predicted without performing the fault injection experiments [64]. The propagation has been examined at the static instruction level, the control flow level and the memory level. For instance, TRIDENT has been used with selective instruction duplication and the experiments show that it can be reduce the SDC probability. The TRIDENT model is improved to predict the SDC of the application by using multiple inputs in another work [65]. TRIDENT is designed for CPU programs, and it cannot be employed with GPU programs because of their high degree of parallelism and different memory architectures. Therefore, the GPU-Trident model is proposed to utilize to predict SDC of GPU application [66]. The SpotSDC is another framework that visualizes the SDC in HPC applications [67]. GPU program's execution characteristic is repetitive and varies by runtime, and the impact of its runtime behavior on the error resiliency is studied [68]. The study also proposed the Spot-FI framework to accelerate the error injecting experiments by finding more vulnerable locations.

Our work differs from these studies in that we carry out a detailed propagation analysis of errors in application data structure space as well as hardware data cache space. Further, we also study the correlation between the propagation in data space and propagation in cache space using two different applications.

3 Error propagation metrics

At a high-level, error propagation can be examined from two different angles. Firstly, it is important to understand the error propagation from an application's perspective. The important questions of interest in this context include: *how quickly is error propagated at an application level*, and *how quickly does an error spread to*

the other data elements when the execution starts with a faulty data element. Secondly, it is important to know how the error propagates across the storage elements, e.g., cache locations. Knowing where the error will be in the cache/memory at any given point of the execution can potentially help us to make more accurate assessments regarding the reliability of a given system.

In this paper, we study the dynamic behavior of error propagation in *both software and hardware levels*, where two metrics, *Propagation of Error on Data Structure* and *Propagation of Error on Hardware*, are proposed to evaluate the error propagation at the software and hardware levels, respectively. The error propagation at the software level is basically the propagation of the error on the application data. On the other hand, error propagation on a cache is considered the error propagation at the hardware layer (though our approach can be extended to other hardware components as well). Caches (especially last-level caches) occupy the majority of chip area in many architectures and have high transistor densities, thus they constitute a suitable medium for the propagation of soft errors, leading to high Soft Error Rates (SERs). In our study, the hardware-level error propagation is examined for caches only; also common in prior works [69–73]. We characterize the error propagation in terms of program points and execution cycles by using our proposed metrics.

3.1 Error Propagation on Data Structures

Our first proposed metric, *Propagation of Error on Data Structure* (***PoEDS***) is defined as corrupted portion of total data per time unit. It measures the error propagation at the software level. The PoEDS has two important advantages. First, since it tracks error propagation during runtime, it gives information about the dynamic behavior of the error. While analyzing the error propagation, we believe it is important to know the dynamic behavior of the error, i.e., which portions of the data elements the error spreads, which pattern the error follows while propagating, or how fast it spreads. Also, one may be interested in exploring the differences in error propagation across different applications.

The user can utilize the fault tolerance mechanisms more efficiently by using this information. To be more concrete, let us assume that there are two different scenarios – in the first one, the error propagates rapidly at the beginning of the execution; but in the second one, the error propagation begins when the execution about to finish. If one could know (or more realistically predict) these error propagation patterns, they could potentially exercise available error protection and corrections mechanisms more judiciously, thereby minimizing the overheads incurred by them. Returning to the two scenarios mentioned above, while utilizing the fault tolerance techniques is necessary from the beginning of the execution in the first scenario, in the second scenario, they could be activated towards the end of the execution. As a result, the overall cost/overhead required by fault tolerance could be minimized in the second scenario.

The second advantage of the PoEDS is that it allows a fine-grain tracking/analysis of errors, where it can examine, for example, the error propagation on a data element caused by another faulty data element. Therefore, an error propagation analysis can

be performed at a data element level by using our proposed metric. Note that such fine-grain analysis is important to ensure *just enough reliability* for several applications where some data may be more critical than others and one has limited execution cycles/power budget that can be allocated for error protection.[2] As an example, in a face recognition application, the part of the image that contains a face can be much more critical than the other parts. If the system has limited resources that can be allocated for fault tolerance, then the protection would be limited to the face part of the image, thereby saving execution cycles and energy.

3.2 Error propagation on hardware

Our second proposed metric, *Propagation of Error on Hardware* (***PoEH***) is defined as the fraction of corrupted locations in a storage component (such as cache or memory) per time unit. While evaluating the corruption, we compare the blocks with their error-free counterparts; if data in the block is erroneous, the cache block is marked as corrupted, and the PoEH is calculated as the ratio between the corrupted blocks and total blocks. Similar to the PoEDS, the PoEH also provides information about the dynamic behavior of the error, since it monitors the error propagation throughout the runtime. The PoEH assists to employ hardware hardening strategies to restrict the error propagation by providing information about which portions of cache/memory contain error at any given point. In addition to the error propagation on storage elements, the cost of the error is another critical aspect when analyzing the error. We perform a cost analysis to calculate the cost of an error. In this analysis, we estimate the fraction of error-free data lost due to incorrect data.

When there is no free location in the cache where the data can be written, the data must be replaced with another data already in the cache. We can divide these replacements into four classes in terms of errors: (i) correct data can be replaced with correct data, (ii) it can be replaced with erroneous data, (iii) erroneous data can be replaced with correct data, or (iv) it can be replaced with erroneous data. While calculating the cost of error, we monitor the memory replacements, and evaluate the fraction of the replacements between correct data and erroneous data. In other words, we measure the fraction of error-free data lost due to the incorrect data.

3.3 Fault injection framework

In this study, we perform comprehensive fault injecting experiments to calculate and track error propagation on software and hardware layers. Figure 1 represents our fault injection framework, which consists of two phases – *profiling phase* and *tracking phase*. In the profiling phase, firstly the application is analyzed to determine which dynamic instructions can be used for fault injection. Then, the fault generator generates the fault file by using dynamic instructions and the fault parameters. In the (error) tracking phase, the errors are injected to the application with GEMFI [74]

[2] This is the case for example in embedded and mobile systems.

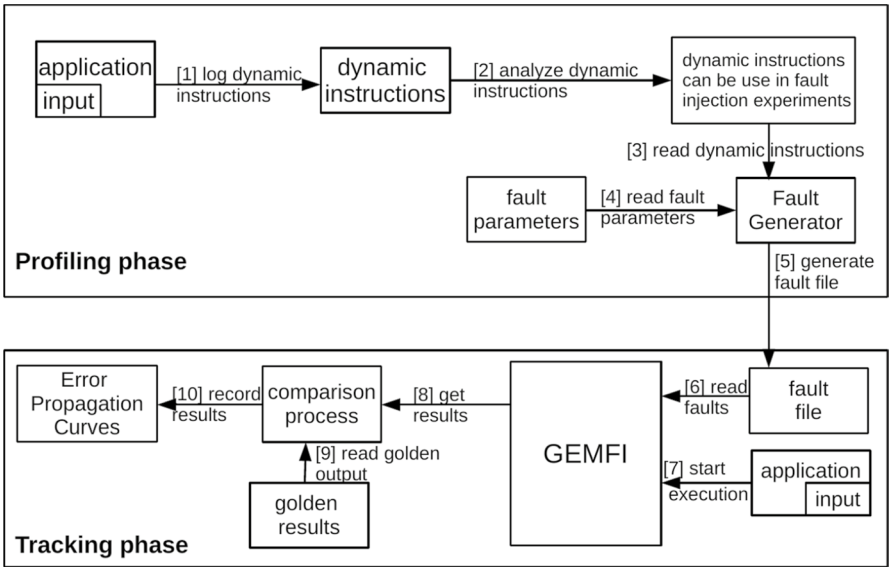


Fig. 1 Fault injection framework

tool. It provides users with an API for fault injection, where the error is introduced to the system at the dynamic instruction level by bit-flip operations. Finally, the results of the corrupted run are compared with the results of the error-free (golden) run.

In the remaining part of this section, fault configuration and parameter details are discussed from the PoEDS perspective. To successfully observe the behavior of the same error at both software and hardware layers, the same error configuration used for calculating both the PoEDS and the PoEH.

Since the PoEDS metric is designed to capture and represent error propagation in and across data structures, the error is introduced into the system by changing the value of the data; specifically, this work focuses on two types of data – **the injected data** and **the monitored data**. While the former represents the data that is injected with one or more errors, the latter represents the data that the error propagates in it. Additionally, the impact of the error rate and the bits affected by the error are also examined by employing four parameters – **erroneous instruction distribution**, **error probability**, **bit-span**, and **bit position**. The erroneous instruction distribution captures the fractions of the instructions that are chosen for error injection, and the error probability represents the probability with which the error injection is performed on the chosen set of instructions. The bit-span captures the number of bits that are affected by the injected errors. Finally, the bit position indicates where the flipped bit is located; e.g., it can be located among the least significant bits or the most significant bits. In our experimental study, a set of errors with various parameters are considered in order to evaluate their impact.

We study the error propagation in the context of *both single-threaded and multi-threaded applications*. Multi-threaded applications have several thread-specific

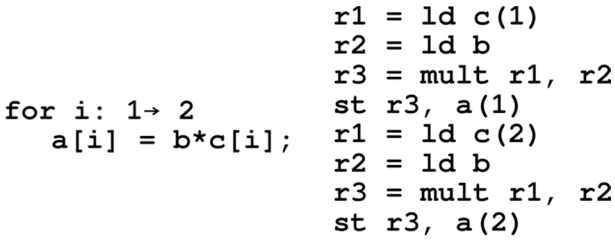


Fig. 2 **(a)** An example code fragment used in our fault injection experiments and **(b)** its corresponding assembly instructions

features. Therefore, besides the common error parameters mentioned above, we employ two additional thread related parameters: **number of threads** and **erroneous thread distribution**. While the number of the threads captures the thread count in the given application, the erroneous thread distribution represents the distribution of threads that contain erroneous data. In this study, we perform experiments with all possible combinations when selecting faulty threads in order to properly examine the effect of the erroneous thread distribution parameter. As an example, if the thread count is equal to 4 where 2 of them contain erroneous data, there would be six different scenarios; thread 0 and thread 1 may be injected, thread 0 and thread 2 may be injected, and so on.

Figure 2 shows a sample code fragment that performs multiplication using array and the corresponding dynamic sequence of assembly instructions generated. Let us assume that this code fragment is provided as an input to our fault injecting framework. In the first step, the monitored and injected data should be decided. In this code fragment, *array c* is the injected data and *array a* is the monitored data. Since our focus is on errors that will cause the **Silent Data Corruption (SDC)** instead of errors that cause system/application hang or abort, the error is injected into the system by altering the value of the elements of *array c* while the error propagation is traced over the elements of the *array a*. In the profiling phase, the **dynamic assembly code** is analyzed. Dynamic code refers to the assembly code generated at runtime for the high-level language. For example, in Fig. 2b, the dynamic instances of c instructions (Fig. 2a) are shown. In Fig. 2, the **ld** instructions read the values of *array c* from the memory and store them in the registers, and **mult** instructions perform multiplication by using these values. Note that, if an error occurs at one of the **ld** or **mult** instructions, that would alter the value of *array c* temporarily. So, for this sample code, the dynamic instruction set for fault injection experiments contains the **ld** and the **mult** instructions. In the tracking phase, the error is injected into the random instructions selected from the instruction set according to the error parameters (erroneous instruction distribution, error probability, bit-span, bit position, number of threads and erroneous thread distribution).

Although the same errors are used in calculating both metrics, tracking error propagation is different for software and hardware layers. The dynamic instruction that caused the error to propagate must be monitored for tracking error

propagation at the software layer. For the given code, **st** instructions write the result of the multiplication into the memory; consequently, the error propagates to arrays via the **st** instructions. To observe error propagation, the **st** instructions from faulty run and error-free run are compared to decide the correctness of *array a* (i.e., the monitored data structure).

On the other hand, since PoEH is defined as the rate of corrupted locations in storage components in the hardware layer, it is necessary to know the contents of the cache and/or memories in order to monitor the propagation of the error. Therefore, when calculating PoEH, the contents of the memory will be detected for faulty run and error-free run at certain CPU cycles, where these snapshots will be compared one another to detect the propagation of the error.

3.3.1 Error model

In this paper, our primary goal is to study soft errors that cause Silent Data Corruption (SDC). Therefore, errors that change the value of the selected data are introduced into the system. The instructions related to the specified data are marked, and errors are injected to their certain bits. Variables such as array indices, loop variables, and errors that can potentially cause a change in the program's control flow are out of the scope of this work. Note also that, since it requires long execution times for an error to disclose itself and our study utilizes benchmarks with not that long execution times, a low error injection rate would cause only a few errors to be injected into the system. In order to present more realistic inference for the impact of soft errors on real-life long-running applications, relatively high error injection rates are used in our experimental study. We want to emphasize that a number of prior studies, including but not limited to literature [36–39], have also adopted such accelerated error injection.

A bit-flip is performed on a randomly chosen single bit, where the bit is chosen among the medium order bits. The double-precision floating-point numbers are utilized in our fault injection experiments, and errors are only injected to medium order mantissa bits. The errors injected into the low-order bits cause small changes that can, in most cases, be ignored. On the other hand, the errors injected into the high-order bits cause significant changes which are easily detectable. Transient single-bit errors are used in fault tolerance studies frequently. So, we have also used this error model in our study. In recent years, multi-bit upsets have also become a major problem; however, we believe single-bit errors are still more prevalent and are a reasonable assumption for applications and target execution environments. Further, it has observed in prior research that single-bit errors can cause as much as SDC rates as double-bit and multi-bit errors (an even higher rates in some cases [75, 76]).

We assume that immediately after the system is started, the application considered in the fault injection experiments is loaded and executed. In other words, the hardware components (memory and cache) are initially empty and error-free. At the beginning of the application execution, all variables are initialized; and the error injection is started after the initialization.

Table 1 The matrices used in the experiments with their dimensions (N) and number of nonzero items (NNZ)

Matrix ID	Matrix	N	NNZ
1	mesh2e1	306	2018
2	mhdb416	416	2312
3	bcsstk07	420	7860
4	nos5	468	5172
5	nos6	675	3255
6	msc00726	726	34518
7	bcsstk09	1083	18437
8	plbuckle	1282	30644
9	mhd4800b	4800	27520
10	fv1	9604	85264

Table 2 Base settings used in the experimental study

Parameter name	Parameter value
(Instruction distribution (%), Error probability)	{(50, 0) (30, 0.03) (20, 0.07)}
Bit span	{single}
Bit position	{medium order}

4 Experimental analysis

The fault injection experiments are performed with the Sparse Matrix Vector Multiplication (SpMV) and the Preconditioned Conjugate Gradient (PCG) using 10 different matrices taken from The University of Florida Sparse Matrix Collection [77]. These matrices and their salient features are given in Table 1.

Table 2 shows the base parameter settings used in our experimental study. The error probability is expected to be low (in practice) for most part of the execution; so, the error rate is kept low for most of the instructions in our experiments. As the value of the error probability parameter is increased, the value of the erroneous instruction rate parameter decreases. The 50% of the dynamic instructions are error-free; for the 30% of the remain instructions, errors are injected with a probability of 0.03; and errors are injected with a probability of 0.07 for the remaining 20% of instructions.

For bit-span parameter, the bit-flip is performed on a randomly chosen single bit, where the bit is chosen from among the medium order bits. The double-precision floating-point numbers represented with 64 bits are used in the experiments. The bits (for injected data) are divided into 3 groups as low-order ($0 - 22$ *bits*), medium-order ($23 - 48$ *bits*), and high-order (*the remaining bits*), where error is injected to the bit selected among medium-order bits, in our experiments.

A sensitivity analysis with different values is performed to examine the impact of each parameter on error propagation separately. Table 3 defines four different scenarios, where each row presents values of parameters for a given scenario.

Table 3 Parameter settings for different experiment scenarios

Experiments	Parameter name	Parameter value
Scenario 1	**(Instruction distribution, Error probability)**	{**(50, 0) (30, 0.03) (20, 0.07), (60, 0) (30, 0.03) (10, 0.07)**}
	Bit span	{single}
Scenario 2	**(Instruction distribution, Error probability)**	{**(50, 0) (30, 0.03) (20, 0.07), (50, 0) (30, 0.1) (20, 0.2)**}
	Bit span	{single}
Scenario 3	(Instruction distribution, Error probability)	{(50, 0) (30, 0.03) (20, 0.07)}
	Bit span	{**single**, **double**}
Scenario 4	**(Instruction distribution, Error probability)**	{**(80, 0) (20, 0.03), (50, 0.1) (30, 0.15) (10, 0.2)**}
	Bit span	{single}

The values given in bold font show alternate values or sets in the experimental study; the default values are given in regular font

Error parameters given in the table are based on the default values. Alternate values are presented for one of the parameters (shown in bold font), and others maintain their default values. Thus, the effect of the parameter on the error propagation is observed more clearly. For example, in the first scenario, the instruction distribution is changed. So the percentage of error-free instruction is increased, and fewer instructions are chosen for fault injection. In the second scenario, the error probability parameter is assumed to have different options. For the third scenario, there are two choices for the bit span parameter: single or double. While single bit means bit-flip operation is executed on one randomly-chosen bit, double bit means bit flip operations are performed on 2 bits (which consists of a randomly-chosen bit and its neighbor). Finally, the fourth scenario is used to observe the system behavior under two opposite cases, namely highly-faulty and low-faulty environments. Since the medium order bits are considered for selecting the candidate bit for all scenarios, bit position is not listed in the table.

For multi-threaded applications, while the values of the base parameters are set based on Table 2, two different alternatives are considered for thread-specific parameters. In the first case, the number of threads is set to 8 or 16 while the erroneous thread distribution is set to 1. On the other hand, the second case considers a fixed number of threads (8), while setting the erroneous thread distribution from the set {1, 4, 8}. Note that the experimental settings presented in this section are used to track both the software-level and the hardware-level error propagation.

5 Results and discussion

In this section, the results of the fault injection experiments are presented in two parts: (a) for software-level error propagation and, (b) for hardware-level error propagation.

5.1 Software-level error propagation

We perform fault injection experiments with different matrices and experimental settings. Only the major computation parts are considered when presenting the results; the initialization parts of the algorithms are not taken into account. For the figures given in this section, the x-axis indicates the time and the y-axis shows the percentage of the corrupted items in the result vector. The minimum and maximum bars are also presented along with the average results. Empirical study spans an extensive evaluation space, since different error configurations, matrices, and applications are considered in the experiments. Similar results are observed from several combinations; therefore, only the results of select combinations are presented in the paper.

The plots in Fig. 3 give the propagation of the errors for the serial SpMV application. SpMV computes $A.x = y$, where A is a sparse matrix. For sparse computations, only non-zero elements and pointer arrays are stored, and the application accesses the elements indirectly (via index arrays). In our experiments, the errors are injected to *matrix A*, and the propagation in the *vector y* is observed. According to the reliability requirements and criticality of data, the injected data and the monitored data can be different. We inject the errors in matrix A, since it is occupied most of the program memory. Vector y is the output of the SpMV kernel, so it is considered as critical data in our study; and error propagation is examined on vector y. The faults in floating-point numbers are determined by comparing erroneous runs with the golden run, where the "absolute difference" between them is computed. If the absolute difference is greater than a certain number, the results are marked as faulty. We consider four significant digits to decide the errors, but it should be noted that the difference may vary depending on the user-level and application-level requirements. Each plot in this figure gives the results using a different matrix and a scenario with different experimental settings. This set of experiments are repeated for all matrices given in Table 3, where Figs. 3a, b, c and d show the propagation results on matrices 1, 5, 6, and 10 given in Table 1.

The experimental results indicate that the error propagation is similar for different matrices and experimental settings. We also note that the error propagation curves shown in Fig. 3 are close to linear, although the propagation speed changes. While the error is propagated to the 20% of the result for one matrix (Fig. 3b), it propagates to the entire result for a different one (Fig. 3c) at the end of the execution. The SpMV algorithm performs a sequence of multiplication operations between vector and matrix elements, where each non-zero element is used once during the iteration. If an error shows up at one of the elements in a row of the matrix, it propagates to

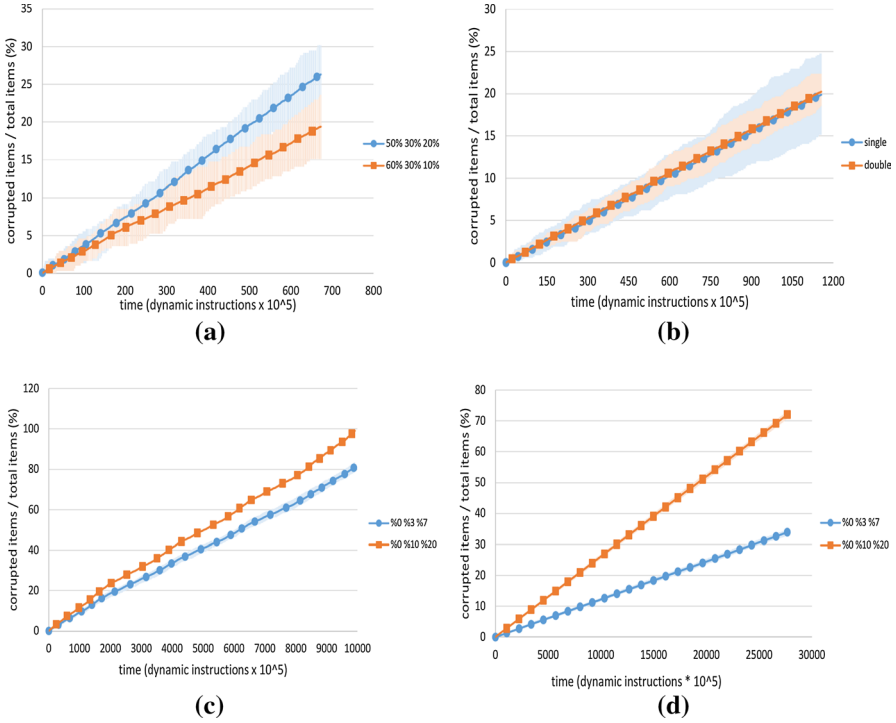


Fig. 3 Error propagation results for serial SpMV application with different matrices and error parameters settings. The matrix ids (from Table 1) and error scenarios (from Table 3) are: (**a**) the first error scenario with matrix 1 (**b**) the third scenario with matrix 5 (**c**) the second scenario with matrix 6, and (**d**) the second scenario with matrix 10

the vector element corresponding to the row, and a repeated pattern of this behavior causes progressive error propagation.

Figure 3a gives the impact of the erroneous instruction distribution based on experimental settings given in Table 3, when the first scenario is considered. The fraction of erroneous instruction decreases, when we consider a lower erroneous instruction distribution, since fewer instructions are injected with errors.

Figure 3b plots the impact of the bit span on the results (the third scenario in Table 3). One can observe that the propagation for different bit span values exhibit similar error propagation patterns. Since the SpMV algorithm performs a series of multiplication operations and the injected errors alter the multiplications results, a flip in one or two bits causes a change in the magnitude of the error (and in both cases it leads to corruption in the same elements).

Fig. 3c and d show the impact of error probability for the second scenario. When the error probability increases, more errors are introduced to the system. As a result, the propagation of the error increases as well. Although the same error probability values are used for both of the experiments, the error propagation is nearly doubled in Fig. 3d, while the difference between the error propagation is smaller in Fig. 3c.

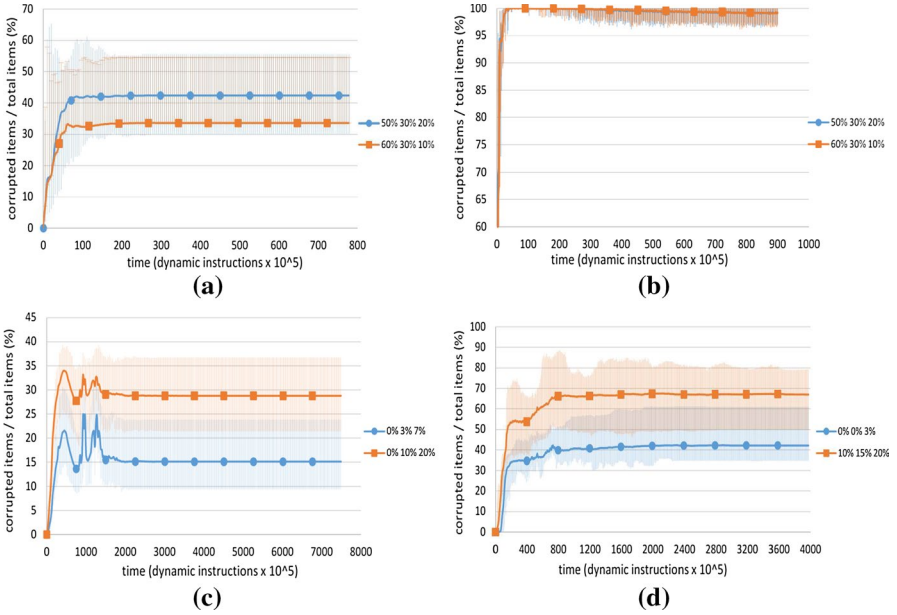


Fig. 4 Error propagation results for serial PCG application with different matrices and error parameters settings. The matrix ids (from Table 1) and error scenarios (from Table 3) are: (**a**) the first error scenario with matrix 1 (**b**) the first error scenario with matrix 2 (**c**) the second error scenario with matrix 6 (**d**) the fourth error scenario with matrix 3

These two graphs show the error propagation on two different matrices with different element distributions and sparsity properties. In Fig. 3c, the ratio between nonzero items and total items for the reference matrix is equal 0.065, whereas the same ratio is equal to 0.0009 in Fig. 3d. Nonzero item distributions and sparsity values of the matrices cause the same error probability values to have different impacts on the error propagation – while the propagation is increased for both the cases, the magnitude of the increase is vastly different.

The graphs in Fig. 4 plot the error propagation when running the serial PCG application, which solves $A.x = y$, where matrix A and the vector y are known. The PCG is an iterative solver that starts its execution with an arbitrary solution (x) and converges to the correct solution at each iteration. Results of the fault injection experiments with PCG are different from the results obtained from SpMV. In contrast to the progressively increasing error propagation curves observed with the SpMV application, error propagation decreases in PCG application at certain periods. It is because of the fact that the algorithm converges the approximate solution as time goes by. When the error occurs, it propagates to the result but eventually the algorithm converges to the right solution despite the error. The divergence-convergence periods can occur more than once during the runtime due to the error injection operation performed at different operating points.

Fig. 4a and b present the impact of the erroneous instruction distribution for the first scenario given in Table 3. When we change the error distribution, the dynamic

instructions that will be injected are increased, which in turn increases the error propagation. As mentioned above, at certain execution periods the error propagation decreases and the algorithm converges to the correct result despite the errors. Although the same error configurations are used for both of the experiments, the error propagates very rapidly in Fig. 4b and almost all items of the result vector become corrupted in a short time. Unlike the graph in Fig. 4a, when the erroneous instruction distribution is changed, the propagation is almost the same for both the high erroneous instruction rate and the low erroneous instruction rate in Fig. 4b. The results of different matrices are given in the two figures (Fig. 4a and b), where different matrices have different effects on the error propagation. Since the fraction of non-zero items in the matrix given in Fig. 4b is very small, flip operations performed on the middle order bits cause a bigger change in the values; therefore, the algorithm could not converge to the correct result. Figure 4c presents the impact of error probability for the second scenario (scenario 2) given in Table 3; and Fig. 4d shows the system behavior in two extreme cases for scenario 4. For both cases, when the error probability increases, more errors are injected to the system, and thus, the error propagation increases, as expected.

Bit position is another parameter employed in our framework. Flipping the most significant bit leads the value to become infinitely large. Therefore, the error propagates to all elements in a short time; and the algorithm could not converge to the correct result. When the least significant bit is flipped, it leads to a negligible change in the values, so the error has almost no effect on the result.

5.1.1 Software-level error propagation in multi-threaded applications

The fault injection experiments are performed by using the parallel version of the iterative SpMV algorithm, where we perform matrix vector multiplication iteratively, $A.x_i = x_{i+1}$. In our implementation, we utilize shared memory communication approach; so, matrix A and vectors (x_i and x_{i+1}) are shared among all threads. The errors are injected to matrix A, and the propagation in the vector x_{i+1} is observed. Figures 5, 6 and 9 plot the results of our experiments (the experimental settings are given in Table 2).

For the iterative SpMV application, the error propagates according to the structure of the sparse matrix [54]. The errors propagate progressively with different patterns for different sparse matrices (see Fig. 5). In this graph, the x-axis is normalized to the range [0..1], in order to fairly compare applications with different execution times. Each run is performed with 8 threads and the errors are injected to the dynamic instructions chosen randomly from all threads.

We consider the erroneous thread distribution parameter in two aspects: *the number of threads that contain errors* and *the thread that is set to be faulty*. In our experiments, these two aspects are taken into consideration individually, and the collected results are presented in Figs. 6 and 9.

The first aspect of the erroneous thread distribution parameter is the number of erroneous threads, where the results are given in Fig. 6a and b. To evaluate the impact of the number of erroneous threads, we inject the same amount of error to

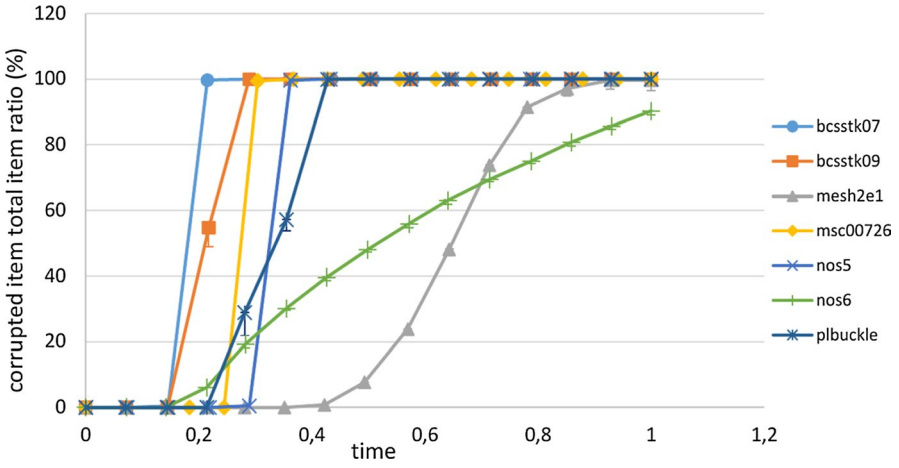


Fig. 5 Error propagation results for parallel iterative SpMV application with different matrices

different number of threads; we compare the error propagation results when eight threads are corrupted, four threads are corrupted and one thread is corrupted. As can be seen from the figures, when more threads are injected with errors by not changing the number of error, error propagation increases.

A simplified scenario is illustrated in Fig. 7 in order to explain this behavior. An almost-diagonal matrix is used, and thread per row approach is employed while sharing the workload among threads. The application is run with five threads, where the matrix and vectors are shared among all threads, and each thread calculates one element of the result vector by using one row. Assume that we consider two different cases when injecting error. Two dynamic instructions are selected from thread 0 in the first case; whereas in the second case, one of the dynamic instructions is selected from thread 0 and the second instruction is selected from thread 2. In the first case, errors are injected to line 3 and line 6 in thread 0. They cause a corruption for the result of the multiplication, and the errors propagate to $y1$ through the computations performed. The propagation of the errors between threads is given in Fig. 8a. An error is injected to line 3 in

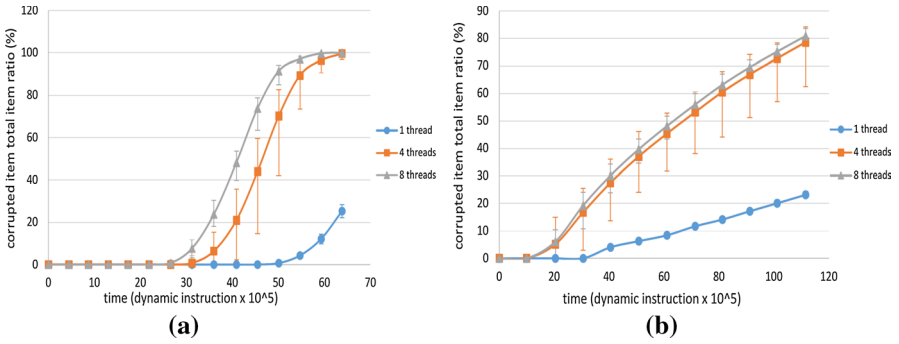
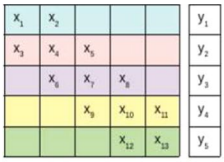


Fig. 6 Error propagation results for parallel iterative SpMV application with different corrupted threads counts and matrices. (**a**) shows results for matrix 1 and (**b**) shows results for matrix 5



Thread 0

```
...
r1 = ld x1
r2 = ld y1
r3 = mult r1,r2
r1 = ld x2
r2 = ld y2
r4 = mult r1,r2
r5 = add r3,r4
wait


st r5, y1
```

Thread 1

```
...
r1 = ld x3
r2 = ld y1
r3 = mult r1,r2
r1 = ld x4
r2 = ld y2
r4 = mult r1,r2
r1 = ld x5
r2 = ld y3
r5 = mult r1,r2
r4 = add r3,r4
r5 = add r4,r5
wait
st r5, y2
```

Thread 2

```
...
r1 = ld x6
r2 = ld y2
r3 = mult r1,r2
r1 = ld x7
r2 = ld y3
r4 = mult r1,r2
r1 = ld x8
r2 = ld y4
r5 = mult r1,r2
r4 = add r3,r4
r5 = add r4,r5
wait
st r5, y3
```

Thread 3

```
...
r1 = ld x9
r2 = ld y3
r3 = mult r1,r2
r1 = ld x10
r2 = ld y4
r4 = mult r1,r2
r1 = ld x11
r2 = ld y5
r5 = mult r1,r2
r4 = add r3,r4
r5 = add r4,r5
wait
st r5, y4
```

Thread 4

```
...
r1 = ld x12
r2 = ld y4
r3 = mult r1,r2
r1 = ld x13
r2 = ld y5
r4 = mult r1,r2
r5 = add r3,r4
wait


st r5, y5
```

Fig. 7 A simplified code and data distribution among threads for the multi-threaded iterative SpMV implementation

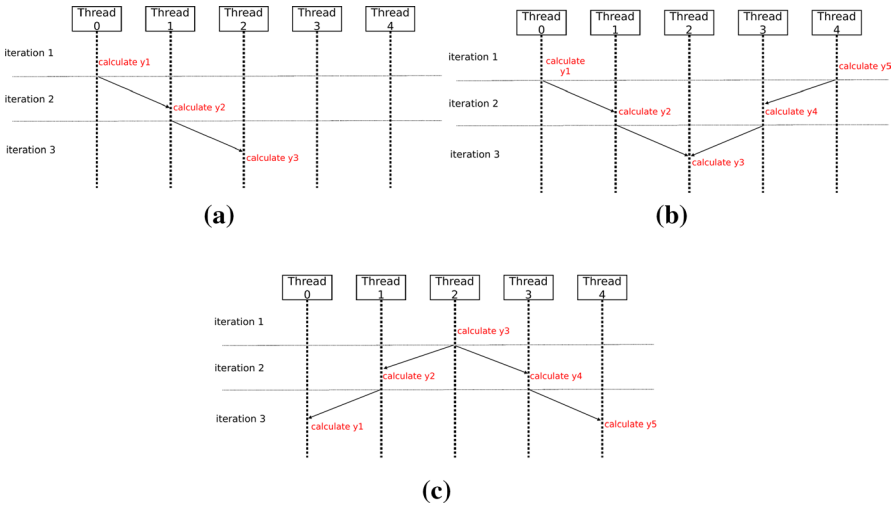


Fig. 8 Error propagation between threads originating from different threads in the multi-threaded iterative SpMV application. Errors are originated from Thread 0 in (**a**), from Threads 0 and 4 in (**b**) and from Thread 2 in (**c**)

thread 0 and an error is injected to line 3 in thread 4 for the second case, where the propagation of the errors between threads is given in Fig. 8b. Although two instructions are injected in both scenarios, the errors are propagated to the entire result vector in the first case, while they are propagated to three data in the second case.

More generally, when we inject the error to threads, it can cause one or more variables that are computed by this thread to become corrupted. Therefore, when

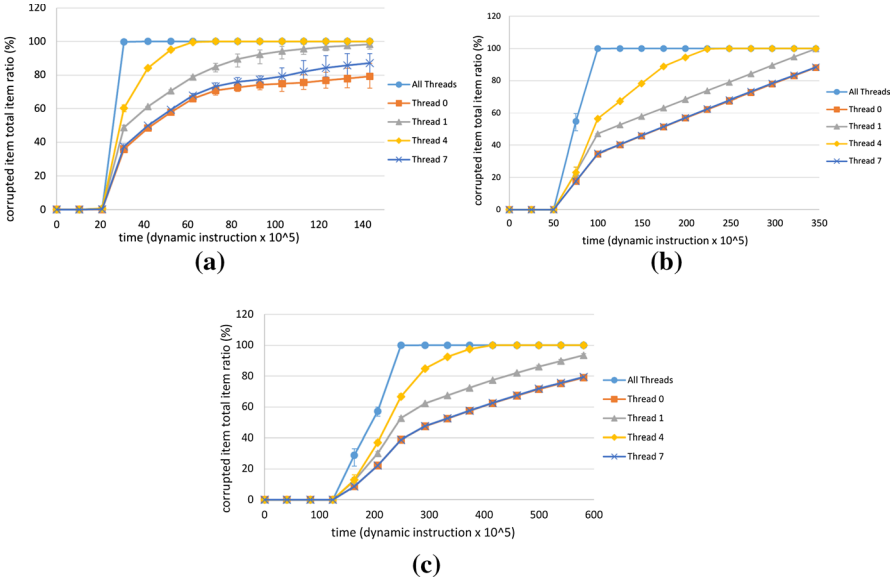


Fig. 9 Error propagation results for the multi-threaded iterative SpMV application with different number of corrupted threads. Each sub-figure shows results for a different matrix: (**a**) matrix 3 (**b**) matrix 7 and (**c**) matrix 8

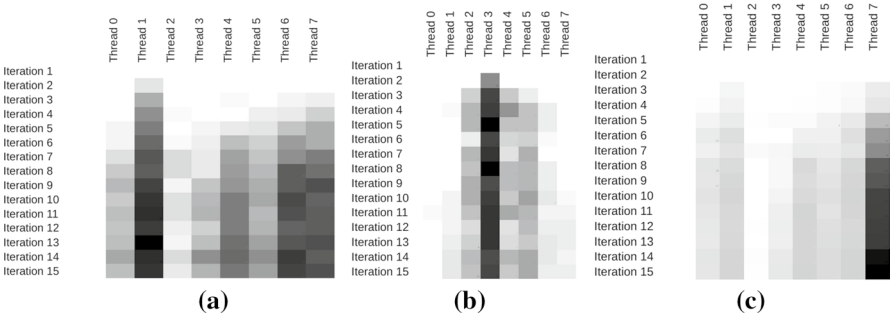


Fig. 10 Error propagation originating from different threads in the multi-threaded iterative SpMV application. The errors are injected to a different thread in each sub-figure (**a**) thread 1, (**b**) thread 3 and (**c**) thread 7

the number of erroneous threads is increased, the probability of corruption for more data arises; and consequently, an increase in error propagation is observed.

For the multi-threaded applications, it is also important to analyze which thread has more impact on error propagation when error is injected to it. In the experiments, the same amount of errors are injected to a single thread, but the thread is chosen differently, where they are presented in Fig. 9a–c. Results show that the error propagation is different when the same amount of errors are injected to the different threads. When an error is injected to a thread, the error is first

Table 4 Parameter settings for different experiment sets

Instruction class	Parameter name	Parameter value
ALU	(Instruction distribution, Error probability)	{(60, 0) (30, 0.03) (10, 0.07)}
	Bit span	{single}
MEM	(Instruction distribution, Error probability)	{(60, 0) (30, 0.03) (10, 0.07)}
	Bit Span	{single}
ALU and MEM	(Instruction distribution, Error probability)	{(60, 0) (30, 0.03) (10, 0.07)}
	Bit span	{single}
ALU and MEM	(Instruction distribution, Error probability)	{(60, 0.1) (30, 0.15) (10, 0.2)}
	Bit span	{single}

propagated within that thread, and then it is propagated to other threads through communications (data sharing). If the thread is strongly connected to others (by sharing lots of data with other threads), the error propagates fast. In Fig. 10, the error propagation between threads is given for the iterative SpMV application. In this figure, the x-axis gives the different threads and the y-axis shows the time in terms of iteration count, where the color represents the number of errors observed on each tread. In another words, higher number of errors in threads are represented by darker colors. The errors are injected to a different thread in each case (i.e, they are injected to thread 1, thread 3 and thread 7), where Fig. 10a–c show the error propagation, respectively. Errors are propagated in different ways depending on the relationship between threads.

The error propagation pattern for different threads can be examined in the application given in Fig. 7. If an error is injected to the 3rd line in thread 0, the error spreads to three variables after three iterations. However, if the error is injected to thread 2, after three iterations, the result propagates to all the elements of the vector. The error propagation originating from thread 0 and thread 2 are given in Fig. 8a and c, respectively.

It is also important to focus on the impact of different instruction classes on error propagation. For this set of experiments, dynamic instructions are divided into two classes – **ALU instructions** and **MEM instructions**; and a total of four different sets are considered (see Table 4). The errors are injected to instructions taken from only one class (either ALU or MEM instructions) for the first two sets; whereas, an equal number of instructions are taken from both classes for the remaining ones. Specifically, for the last set, 30% of the dynamic instructions (i.e., 15% of the ALU instructions and 15% of the MEM instructions) are injected with a probability of 0.15. The multi-threaded version of the iterative SpMV algorithm is used for the experiments by running with 8 threads, where the errors are injected to the instructions selected from all threads. The error propagation graphs given in Fig. 11 show similar error propagation patterns across different instruction classes, which is due to the dependencies between the instructions. For the simplified SpMV code given in Fig. 7, the error dependency between dynamic instructions is illustrated in Fig. 12. For this code, an error that

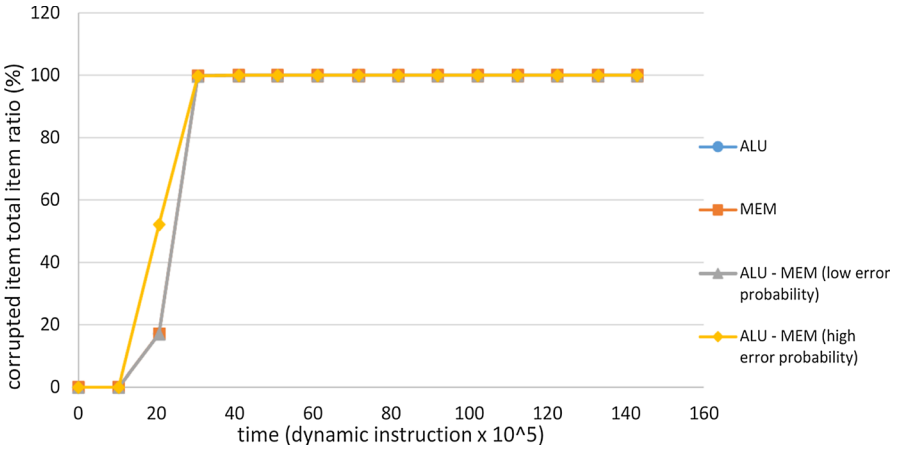


Fig. 11 Error propagation for different instruction types

	ld x1	ld y1	mult r1, r2	ld x2	ld y2	mult r1, r2	add r3, r4	st r5, y1
ld x1	X		X				X	X
ld y1		X	X				X	X
mult r1, r2			X				X	X
ld x2				X		X	X	X
ld y2					X	X	X	X
mult r1, r2						X	X	X
add r3, r4							X	X
st r5, y1								X

Fig. 12 Dependencies between dynamic instructions

occurs in an MEM instruction immediately propagates to an instruction from the ALU class; and likewise, an error originating from an ALU instruction immediately propagates to a MEM instruction. For example, if instruction **ld x1** (a MEM instruction) from thread 0 is corrupted, this error propagates to three more instructions (**mult r1, r2**, **add r3, r4**, and **st r5, y1**) and two of them are the ALU instructions.

5.2 Hardware-level propagation

When examining the error at the hardware level, we consider the error in two dimensions, the error propagation and the error cost, by utilizing the same matrices and error configurations from the previous section. As in Sect. 5.1, major computation

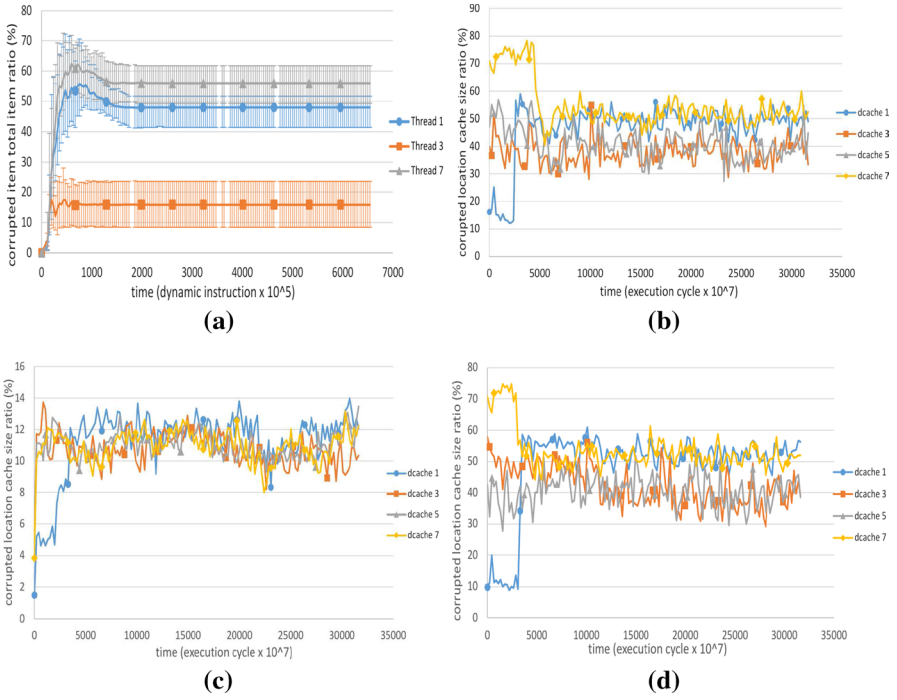


Fig. 13 Error propagation on hardware for the multi-threaded PCG application with different erroneous threads. (**a**) shows error propagation on the software level originating from thread 1, thread 3 and thread 5, and the corresponding error propagation on hardware are shown in (**b**), (**c**) and (**d**), respectively

parts are considered when presenting the results. In the graphs presented below, the y-axis shows the percentage of the erroneous location in the cache, and the x-axis gives the time in terms of execution cycles.

Figure 13 gives error propagation results on different data caches for the PCG algorithm. By injecting the error on three different threads (thread 1, thread 3 and thread 7), Fig. 13a shows error propagation on the software level originating from those threads, separately. The corresponding hardware level error propagation of Fig. 13a for the three cases (thread 1, thread 3 and thread 5) are given in Fig. 13b, c and d, respectively. When the errors are injected to thread 1 or thread 7, it propagates to more amount data than the case of injecting on thread 3 (see Fig. 13a). A similar observation is performed for the hardware level. It propagates to 60% of the cache size when thread 1 or thread 7 is injected (see Figs. 13b and d); whereas the ratio is only 14% when thread 3 is considered (Fig. 13c).

On the other hand, the propagation pattern shows a different trend at the hardware level. Specifically, the hardware-level error propagation fluctuates drastically throughout the execution. The reason for this is the cache-specific behavior – since not all data can fit in the cache at any given time, the execution experiences frequent cache misses, which in turn changes the fraction of erroneous data in the cache.

The error propagation results for the iterative SpMV algorithm are slightly different. Figure 14 shows the error propagation on hardware level for iterative SpMV

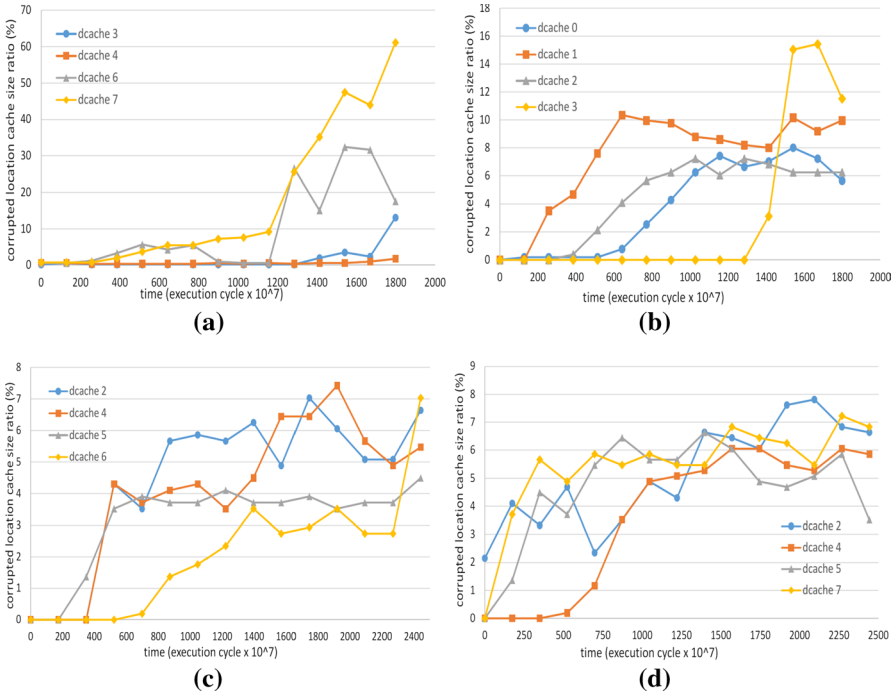


Fig. 14 Error propagation on hardware for the multi-threaded iterative SpMV application with different erroneous threads. For the first two sub-figures errors are injected to one thread: (**a**) thread 0 and (**b**) thread 3. For the later two sub-figures, errors are injected to two threads: (**c**) threads 0 and 4 and (**d**) threads 4 and 6

algorithm in a multi-core system. In order to observe the error propagation in different data caches, different combinations of data caches are presented in the graphs. As can be seen from the figure, there are fewer fluctuations in error propagation curves and the errors propagate progressively. The reason for the difference is that iterative SpMV algorithm utilizes fewer data structures, so the data may fit into the caches more easily and cache replacements occur less frequently.

To understand the different behaviour of data propagation at hardware and software levels more clearly, we discuss how the error propagates between instructions and the caches. In our study, we consider the errors that will cause (Silent Data Corruption) SDC by altering the value of the data such as incorrect calculation due to the error in the ALU, or error in the data during writing from or reading to memory (load/store operations). A bit flip occurs in the data bits of the dynamic instruction spreads to other dynamic instructions via data dependency, which causes propagation and increase in data corruption.

The propagation of the error to cache or memory occurs via load instructions, when faulty data is written to the memory elements. As an example, the code block given in Fig. 15a is taken from the iterative SpMV application. In the code, elements of the vector and matrix are read from the memory with load (*ldfp*)

```
ld eax, DS [rbp + 0xfffffffffffffe8]          (eax = 0x0000000C00000005)
ldfp %xmm1_low, DS [8*rax + rdx]              (%xmm1_low = 0x3ff0000000000000)
ld eax, DS [rbp + 0xfffffffffffffe8]          (eax = 0x0000000C00000005)
ldfp %xmm0_low, DS [8*rax + rdx]              (%xmm0_low = 0x4C41000000000000)
mmulf %xmm1_low, %xmm1_low, %xmm0_low         (%xmm1_low = 0x4C41000000000000)
ldfp %xmm0_low, DS [rbp + 0xffffffffffffff0] (%xmm0_low = 0x4C4a801800000000)
maddf %xmm0_low, %xmm0_low, %xmm1_low         (%xmm0_low = 0x4055c00c00000000)
stfp %xmm0_low, DS [rbp + 0xffffffffffffff0] (store 0x4055c00c00000000)
```

(a)

tag	data	tag	data
ff03f	0x4055c00c00000000	8b39	0x860000007e
ff03f	0x4032800000000000	8b3b	0x3ff0000000000000
ff03f	0x4041000000000000	8b3b	0x3ff0000000000000
8b38	0x900000007	8b3a	0x3ff0000000000000
8b3b	0x7fffa6a39e00	ff03f	0x408f7c0000000000
…	…	…	…
…	…	…	…
…	…	…	…

(b)

```
ld eax, DS [rbp + 0xfffffffffffffe8]          (eax = 0x0000000000000000)
ldfp %xmm1_low, DS [8*rax + rdx]              (%xmm1_low = 0x4008000000000000)
ld eax, DS [rbp + 0xfffffffffffffe8]          (eax = 0x0000000000000000)
ldfp %xmm0_low, DS [8*rax + rdx]              (%xmm0_low = 0x4055c00c00000000)
mmulf %xmm1_low, %xmm1_low, %xmm0_low         (%xmm1_low = 0x4070500900000000)
ldfp %xmm0_low, DS [rbp + 0xffffffffffffff0] (%xmm0_low = 0x0000000000000000)
maddf %xmm0_low, %xmm0_low, %xmm1_low         (%xmm0_low = 0x4070500900000000)
stfp %xmm0_low, DS [rbp + 0xffffffffffffff0] (store 0x4070500900000000)
```

(c)

Fig. 15 Propagation of an error observed at an ALU operation in the cache: (**a**) a code block taken from the iterative SpMV application where erroneous instructions are represented with red. (**b**) the status of the cache after "(a)" is executed (**c**) code block read erroneous data from the cache

instructions; and calculations are performed using ALU instructions (*mmulf*, *maddf*). When an error occurs in the ALU while it performs the *maddf* instruction, the result of the *maddf* instruction is calculated incorrectly and the erroneous data is written into the memory using the store instruction (*stfp*). Thus, the error in the registers propagates to the memory. Figure 15b shows the status of the cache after the code block is executed. The erroneous data in the cache are read in later iterations via the load instruction, and they are used in subsequent ALU operations which cause the continuation of error propagate. As can be seen

```
ld eax, DS [rbp + 0xfffffffffffffe8]          (eax = 0x000000000000002c)
ldfp %xmm1_low, DS [8*rax + rdx]              (%xmm1_low = 0x3ff0000000000000)
ld eax, DS [rbp + 0xfffffffffffffe8]          (eax = 0x000000000000002c)
ldfp %xmm0_low, DS [8*rax + rdx]              (%xmm0_low = 0x4062d00200000000)
mmulf %xmm1_low, %xmm1_low, %xmm0_low         (%xmm1_low = 0x4062d00200000000)
ldfp %xmm0_low, DS [rbp + 0xffffffffffffff0] (%xmm0_low = 0x0000000000000000)
maddf %xmm0_low, %xmm0_low, %xmm1_low         (%xmm0_low = 0x4062d00200000000)
stfp %xmm0_low, DS [rbp + 0xffffffffffffff0] (store 0x4062d00200000000)
```

(a)

tag	data	tag	data
ff03f	0x4010004000000000	8b39	0x3ff0000000000000
ff03f	0x4032800000000000	8b3b	0x3ff0000000000000
ff03f	0x900000008	8b3b	0x4041000000000000
8b38	0x3ff0000000000000	8b3a	0x4062d002
8b3b	0x7fffa6a39e00	ff03f	0x408f7c0000000000
…	…	…	…
…	…	…	…
…	…	…	…

(b)

```
ld eax, DS [rbp + 0xfffffffffffffe8]          (eax = 0x000000000000002d)
ldfp %xmm1_low, DS [8*rax + rdx]              (%xmm1_low = 0x3ff0000000000000)
ld eax, DS [rbp + 0xfffffffffffffe8]          (eax = 0x000000000000002d)
```

(c)

tag	data	tag	data
ff03f	0x4010004000000000	8b39	0x3ff0000000000000
ff03f	0x4032800000000000	8b3b	0x3ff0000000000000
ff03f	0x900000008	8b3b	0x4041000000000000
8b38	0x3ff0000000000000	8b3a	0x3ff0000000000000
8b3b	0x7fffa6a39e00	ff03f	0x408f7c0000000000
…	…	…	…
…	…	…	…
…	…	…	…

(d)

Fig. 16 Replacement of erroneous code block in the cache. (**a**) a code block taken from the iterative SpMV application where erroneous instructions are represented with red. (**b**) the status of the cache after "(a)" is executed. (**c**) code block read erroneous data from the cache shown in "(b)". (**d**) the final status of cache

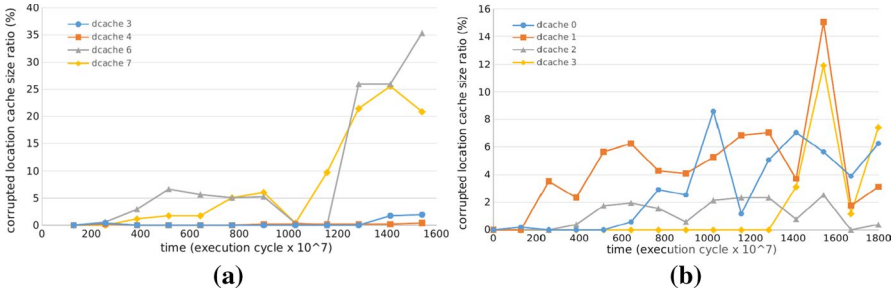


Fig. 17 Cost analysis for errors in iterative SpMV application at hardware level with different erroneous threads. Errors are injected to thread 0 (**a**) and thread 3 (**b**)

from Fig. 15c, the erroneous data are read via load and they are used as input in ALU operations.

The error propagated to the cache (i.e., the erroneous data in the cache) might be replaced before it is used. While the software level error has more stable propagation, this cache specific behavior causes fluctuations in the error propagation curves for the cache. An example scenario for cache replacement is given in Fig. 16. Erroneous data is written to the memory through the store commands, causing an increase in the error rate in the cache. After the first part of the code block in Fig. 16a is executed, there will be two erroneous blocks in the cache (see Fig. 16b). When the following instructions (Fig. 16c) are executed, the erroneous code blocks are replaced with an error-free data; and the number of incorrect blocks decreases to one (Fig. 16d).

It is also important to measure the cost of the error in terms of the displacement of error-free data in the caches. For this cost analysis, *error costs* are calculated as *the portion of erroneous data replaced by error-free data* in Fig. 17. The error cost graph in Fig. 17a corresponds to the error propagation in Fig. 14a; and Fig. 17b corresponds to the error propagation in Fig. 14b. Although the error cost graphs show similar patterns to the corresponding error propagation graphs, there are more fluctuations than the error propagation curves. Moreover, the error cost is lower than the error propagation. It is due to the fact that the erroneous data are overwritten with erroneous data occasionally during the execution; as a result, error propagation may not be always reflected in the cost directly.

5.3 Correlation between hardware and software level propagation

Up to now, we analyze error propagation patterns in the software (application) layer and hardware layer, separately. Another extension is to study the correlation between these two propagation. To evaluate the correlation between the EPoDS and the EPoH for the iterative SpMV application, we calculate the *Pearson* and *Spearman* correlation coefficients. Both correlation coefficients take values between $+1$ and -1, where $+1$ indicates a positive correlation, -1 a negative correlation, and 0 no correlation. It should be noted that while the Pearson's correlation coefficient refers

Table 5 Pearson and Spearman correlation coefficients between software and hardware level error propagation for the iterative SpMV application with different matrices

Matrix	Pearson's correlation Coefficient	Spearman's correlation Coefficient
mesh2e1	0.80	0.82
bcsstk07	0.91	0.76
nos5	0.85	0.85
nos6	0.81	0.89
msc00726	0.99	0.62
bcsstk09	0.89	0.67
plbuckle	0.90	0.85
mhd4800b	0.32	0.69
fv1	0.50	0.47

Table 6 Pearson and Spearman correlation coefficients for the iterative SpMV application on different data cache sizes

Matrix	32kB Data cache		128kB Data cache	
	Pearson's correlation	Spearman's correlation	Pearson's correlation	Spearman's correlation
nos6	0.806	0.892	0.802	0.857
plbuckle	0.904	0.846	0.908	0.846
fv1	0.495	0.469	0.57	0.887

to a linear relationship, the Spearman's correlation coefficient refers to a monotonic relationship.

Table 5 presents the correlation coefficient values between error propagation in data caches and error propagation at the software level for the iterative SpMV application using different matrices. Correlation coefficients are generally close to +1, indicating that there is a strong correlation between error propagation at the software and hardware layers. On the other hand, the correlation coefficients for the matrices given in the last two rows of the table are significantly smaller than others, which indicates weaker relationship. One possible reason is that for these cases the total size of matrices is larger than the other cases. The increase in the size of the data used by the application causes the data not being able to fit in the cache; and eventually replacements are performed more frequently. In some cases, the erroneous data are replaced with the new ones; therefore, the erroneous corruption rate in the cache decreases, while the corrupted data at the software level still increases.

If we repeat the experiments by increasing the size of the data cache, the correlation coefficients increase. Table 6 shows the Pearson's and Spearman's correlation coefficients between software and hardware level error propagation for the iterative SpMV application using different matrices and data cache sizes. For the matrices *nos*6 and *plbuckle*, the correlation coefficient is close to +1 for 32kB size data cache, and effect of bigger data cache size is negligible. However, the correlation coefficient increases, when we increase the data cache size for *fv*1 matrix.

6 Concluding remarks

In this paper, we analyze the soft error propagation characteristics at the hardware and software levels. We propose two metrics: a) Propagation of Error on Data Structure (PoEDS) for tracking the error propagation at the software level, and b) Propagation of Error on Hardware (PoEH) for tracking the error propagation at the hardware level. Our proposed metrics are designed to monitor the error propagation during the execution, and therefore, they can be used to capture information about the dynamic behavior of the error.

To evaluate the proposed metrics thoroughly, we perform various fault injection experiments with two different single-thread and multi-threaded applications and observe the impact of different factors on the error propagation. Our experimental study reveals that the error propagation is highly dependent on the types of computations performed in a given program as well as the order at which they are performed. The error propagation generally increases by time but, for some kernels, the application can produce correct results despite the error. Also, the error propagation pattern can change according to the input.

The experimental study demonstrates that the error propagation have different characteristics for hardware and software levels. At the software level, the error propagates more smoothly. However, at the hardware level, the error propagation curves exhibit more severe fluctuations. The reason behind that, since the cache size is limited only a part of the erroneous data can be located in it. Consequently, even though the number of erroneous data increases over time, it is not always reflected in the corrupted cache locations. After the error propagation at software and hardware level are studied separately, the correlation between them is also examined. The Pearson's and Spearman's correlation coefficients reveal that there is a positive relation between the hardware and software level error propagation.

We anticipate two major extensions for this work – the first one is on further exploring the relationship between the hardware level and software level error propagation. The second possible extension is to slow down the error propagation. If one could develop an error prediction strategy based on our characterization findings, it may be possible to make dynamic changes to the running application code and/or execution environment to reduce faults.

Acknowledgements This research was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) with a research grant (Project Number: 118E715).

Data Availability The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

References

1. Rebaudengo M, Reorda MS, Violante M. An accurate analysis of the effects of soft errors in the instruction and data caches of a pipelined microprocessor. In (2003) Design. Autom Test Europe Conf Exhib 2003:602–607

2. Gold BT, Smolens JC, Falsafi B, Hoe JC. The granularity of soft-error containment in shared-memory multiprocessors; 2006
3. Smolens JC, Gold BT, Kim J, Falsafi B, Hoe JC, Nowatzyk AG. Fingerprinting: bounding soft-error detection latency and bandwidth. In: ACM SIGPLAN Notices. vol. 39; 2004. p. 224–234
4. Medeiros GE, Bortolon FT, Reis R, Ost L. Evaluation of compiler optimization flags effects on soft error resiliency. In: 2018 31st Symposium on Integrated Circuits and Systems Design (SBCCI); 2018. p. 1–6
5. Gava J, Bandiera V, Reis R, Ost L. Evaluation of compilers effects on OpenMP soft error resiliency. In: 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI); 2019. p. 259–264
6. Lins FM, Tambara LA, Kastensmidt FL, Rech P (2017) Register file criticality and compiler optimization effects on embedded microprocessor reliability. IEEE Trans Nucl Sci 64(8):2179–2187
7. Baumann RC. Radiation-induced soft errors in advanced semiconductor technologies. IEEE Transactions on Device and Materials Reliability. 2005 Sept;5(3):305–316
8. Cappello F, Al G, Gropp W, Kale S, Kramer B, Snir M (2014) Toward exascale resilience: 2014 update. Supercomput Frontiers Innovations: Int J 1(1):5–28
9. Baumann RC (2001) Soft errors in advanced semiconductor devices-part I: the three radiation sources. IEEE Trans Device Mater Reliab 1(1):17–22
10. Shivakumar P, Kistler M, Keckler SW, Burger D, Alvisi L. Modeling the effect of technology trends on the soft error rate of combinational logic. In: Proceedings International Conference on Dependable Systems and Networks; 2002. p. 389–398
11. O'Gorman TJ, Ross JM, Taber AH, Ziegler JF, Muhlfeld HP, Montrose CJ et al (1996) Field testing for cosmic ray soft errors in semiconductor memories. IBM J Res Dev 40(1):41–50
12. Borkar S (2005) Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro 25(6):10–16
13. Asadi GH, Sridharan V, Tahoori MB, Kaeli D. Balancing performance and reliability in the memory hierarchy. In: IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.; 2005. p. 269–279
14. Sikai L, Jun Y. A method of soft error propagation based on cellular automata. In: 2018 IEEE 3rd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA); 2018. p. 617–622
15. Mukherjee SS, Kontz M, Reinhardt SK. Detailed design and evaluation of redundant multi-threading alternatives. In: Proceedings 29th Annual International Symposium on Computer Architecture; 2002. p. 99–110
16. Reinhardt SK, Mukherjee SS. Transient fault detection via simultaneous multithreading. In: Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201); 2000. p. 25–36
17. Rotta R, Ferreira RS, Nolte J. Real-time dynamic hardware reconfiguration for processors with redundant functional units. In: 2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC); 2020. p. 154–155
18. Ainsworth S, Jones TM. Parallel error detection using heterogeneous cores. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN); 2018. p. 338–349
19. Györök G, Beszédes B. Duplicated control unit based embedded fault-masking systems. In: 2017 IEEE 15th International Symposium on Intelligent Systems and Informatics (SISY); 2017. p. 283–288
20. Reis GA, Chang J, Vachharajani N, Rangan R, August DI. SWIFT: Software implemented fault tolerance. In: Proceedings of the International Symposium on Code Generation and Optimization; 2005. p. 243–254
21. Asghari SA, Marvasti MB, Rahmani AM (2018) Enhancing transient fault tolerance in embedded systems through an OS task level redundancy approach. Futur Gener Comput Syst 87:58–65
22. Mahmoud A, Hari SKS, Sullivan MB, Tsai T, Keckler SW. Optimizing software-directed instruction replication for gpu error detection. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis; 2018. p. 842–853
23. Thati VB, Vankeirsbilck J, Penneman N, Pissoort D, Boydens J. An improved data error detection technique for dependable embedded software. In: 2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC); 2018. p. 213–220

24. Chen YS, Chen PS. A software-based redundant execution programming model for transient fault detection and correction. In: 2016 45th International Conference on Parallel Processing Workshops (ICPPW); 2016. p. 66–71
25. Vallero A, Savino A, Chatzidimitriou A, Kaliorakis M, Kooli M, Riera M et al (2018) SyRA: Early system reliability analysis for cross-layer soft errors resilience in memory arrays of microprocessor systems. IEEE Trans Comput 68(5):765–783
26. Cheng E, Mirkhani S, Szafaryn LG, Cher CY, Cho H, Skadron K, et al. CLEAR: Cross-layer exploration for architecting resilience-combining hardware and software techniques to tolerate soft errors in processor cores. In: Proceedings of the 53rd Annual Design Automation Conference; 2016. p. 1–6
27. Vallero A, Savino A, Politano G, Di Carlo S, Chatzidimitriou A, Tselonis S, et al. Cross-layer system reliability assessment framework for hardware faults. In: 2016 IEEE International Test Conference (ITC); 2016. p. 1–10
28. Gupta M, Sridharan V, Roberts D, Prodromou A, Venkat A, Tullsen D, et al. Reliability-aware data placement for heterogeneous memory architecture. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA); 2018. p. 583–595
29. Jaulmes L, Moretó M, Valero M, Erez M, Casas M. Runtime-guided ECC protection using online estimation of memory vulnerability. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis; 2020. p. 1–14
30. Asadi G, Tahoori MB. An analytical approach for soft error rate estimation in digital circuits. In: 2005 IEEE International Symposium on Circuits and Systems; 2005. p. 2991–2994
31. Mukherjee SS, Emer J, Reinhardt SK. The soft error problem: An architectural perspective. In: 11th International Symposium on High-Performance Computer Architecture; 2005. p. 243–247
32. Weaver C, Emer J, Mukherjee SS, Reinhardt SK (2004) Techniques to reduce the soft error rate of a high-performance microprocessor. ACM SIGARCH Comput Archit News 32(2):264
33. Upasani G, Vera X, González A. Reducing due-fit of caches by exploiting acoustic wave detectors for error recovery. In: 2013 IEEE 19th International On-Line Testing Symposium (IOLTS); 2013. p. 85–91
34. Fratin V, Oliveira D, Lunardi C, Santos F, Rodrigues G, Rech P. Code-dependent and architecture-dependent reliability behaviors. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN); 2018. p. 13–26
35. Utrera G, Gil M, Martorell X. Analysis of the impact factors on data error propagation in HPC applications. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP); 2018. p. 546–549
36. Ferreira RR, Da Rolt J, Nazar GL, Moreira AF, Carro L. Adaptive low-power architecture for high-performance and reliable embedded computing. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks; 2014. p. 538–549
37. Hu J, Wang S, Ziavras SG. On the exploitation of narrow-width values for improving register file reliability. IEEE Transactions on Very Large Scale Integration (VLSI) systems. 2009;17(7):953–963
38. Subasi O, Arias J, Unsal O, Labarta J, Cristal A. Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing; 2015. p. 99–102
39. Ashraf RA, Gioiosa R, Kestor G, DeMara RF. Exploring the effect of compiler optimizations on the reliability of HPC applications. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); 2017. p. 1274–1283
40. Mukherjee SS, Weaver C, Emer J, Reinhardt SK, Austin T. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.; 2003. p. 29–40
41. Mukherjee SS, Weaver CT, Emer J, Reinhardt SK, Austin T (2003) Measuring architectural vulnerability factors. IEEE Micro 23(6):70–75
42. Zhang W. Computing cache vulnerability to transient errors and its implication. In: 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05); 2005. p. 427–435
43. Yan J, Zhang W. Compiler-guided register reliability improvement against soft errors. In: Proceedings of the 5th ACM International Conference on Embedded Software; 2005. p. 203–209
44. Jaulmes L, Moreto M, Valero M, Casas M. A vulnerability factor for ECC-protected memory. In: 2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS); 2019. p. 176–181

45. Sridharan V, Kaeli DR. Eliminating microarchitectural dependency from architectural vulnerability. In: 2009 IEEE 15th International Symposium on High Performance Computer Architecture; 2009. p. 117–128
46. Borodin D, Juurlink BH. Protective redundancy overhead reduction using instruction vulnerability factor. In: Proceedings of the 7th ACM International Conference on Computing Frontiers; 2010. p. 319–326
47. Yu L, Li D, Mittal S, Vetter JS. Quantitatively modeling application resilience with the data vulnerability factor. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2014. p. 695–706
48. Oz I, Topcuoglu HR, Kandemir M, Tosun O (2012) Thread vulnerability in parallel applications. J Parallel Distribut Comput 72(10):1171–1185
49. Hiller M, Jhumka A, Suri N. On the placement of software mechanisms for detection of data errors. In: Proceedings International Conference on Dependable Systems and Networks; 2002. p. 135–144
50. Leeke M, Jhumka A. Towards understanding the importance of variables in dependable software. In: 2010 European Dependable Computing Conference; 2010. p. 85–94
51. Utrera G, Gil M, Martorell X. Analyzing data-error propagation effects in high-performance computing. In: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP); 2016. p. 418–421
52. Ashraf RA, Gioiosa R, Kestor G, DeMara RF, Cher CY, Bose P. Understanding the propagation of transient errors in HPC applications. In: SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2015. p. 1–12
53. Guo L, Li D. Moard: Modeling application resilience to transient faults on data objects. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2019. p. 878–889
54. Shantharam M, Srinivasmurthy S, Raghavan P. Characterizing the impact of soft errors on iterative methods in scientific computing. In: Proceedings of the International Conference on Supercomputing; 2011. p. 152–161
55. Moríñigo JA, Bustos A, Mayo-García R (2022) Error resilience of three GMRES implementations under fault injection. J Supercomput 78(5):7158–7185
56. Guo L, Li D, Laguna I, Schulz M. Fliptracker: Understanding natural error resilience in hpc applications. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis; 2018. p. 94–107
57. Fiala D, Mueller F, Engelmann C, Riesen R, Ferreira K, Brightwell R. Detection and correction of silent data corruption for large-scale high-performance computing. In: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis; 2012. p. 1–12
58. Guan Q, Hu X, Grove T, Fang B, Jiang H, Yin H, et al. Chaser: An enhanced fault injection tool for tracing soft errors in mpi applications. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN); 2020. p. 355–363
59. DeFreez D, Bhowmick A, Laguna I, Rubio-González C. Detecting and reproducing error-code propagation bugs in MPI implementations. In: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming; 2020. p. 187–201
60. Somani AK, Trivedi KS. A cache error propagation model. In: Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems; 1997. p. 15–21
61. Li ML, Ramachandran P, Sahoo SK, Adve SV, Adve VS, Zhou Y (2008) Understanding the propagation of hard errors to software and implications for resilient system design. ACM Sigplan Notice 43(3):265–276
62. Gu J, Zheng W, Zhuang Y, Zhang Q (2019) Vulnerability analysis of instructions for SDC-causing error detection. IEEE Access 7:168885–168898
63. Li Z, Menon H, Mohror K, Bremer PT, Livant Y, Pascucci V. Understanding a program's resiliency through error propagation. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming; 2021. p. 362–373
64. Li G, Pattabiraman K, Hari SKS, Sullivan M, Tsai T. Modeling soft-error propagation in programs. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN); 2018. p. 27–38
65. Li G, Pattabiraman K. Modeling input-dependent error propagation in programs. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN); 2018. p. 279–290

66. Anwer AR, Li G, Pattabiraman K, Sullivan M, Tsai T, Hari SKS. Gpu-trident: efficient modeling of error propagation in gpu programs. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis; 2020. p. 1–15
67. Li Z, Menon H, Maljovec D, Livnat Y, Liu S, Mohror K et al (2020) Spotsdc: Revealing the silent data corruption propagation in high-performance computing systems. IEEE Trans Visual Comput Graph 27(10):3938–3952
68. Previlon F, Kalra C, Tiwari D, Kaeli D (2022) Characterizing and exploiting soft error vulnerability phase behavior in gpu applications. IEEE Trans Dependable Secure Comput 19(1):288–300
69. Ko Y, Jeyapaul R, Kim Y, Lee K, Shrivastava A (2017) Protecting caches from soft errors: a micro-architect's perspective. ACM Trans Embed Comput Sys (TECS) 16(4):1–28
70. Mittal S, Vetter JS. Reducing soft-error vulnerability of caches using data compression. In: Proceedings of the 26th Edition on Great Lakes Symposium on VLSI; 2016. p. 197–202
71. Houssany S, Guibbaud N, Bougerol A, Leveugle R, Miller F, Buard N (2012) Microprocessor soft error rate prediction based on cache memory analysis. IEEE Trans Nucl Sci 59(4):980–987
72. Vijayan A, Koneru A, Ebrahimit M, Chakrabarty K, Tahoori MB. Online soft-error vulnerability estimation for memory arrays. In: 2016 IEEE 34th VLSI Test Symposium (VTS); 2016. p. 1–6
73. Mamoutova OV, Antonov AP, Filippov AS. On design of cache with efficient soft error protection. In: 2017 IEEE 37th International Conference on Electronics and Nanotechnology (ELNANO); 2017. p. 57–60
74. Parasyris K, Tziantzoulis G, Antonopoulos CD, Bellas N. GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks; 2014. p. 622–629
75. Sangchoolie B, Pattabiraman K, Karlsson J. One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors. In: 2017 47th annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN); 2017. p. 97–108
76. Lu Q, Farahani M, Wei J, Thomas A, Pattabiraman K. Llfi: An intermediate code-level fault injection tool for hardware faults. In: 2015 IEEE International Conference on Software Quality, Reliability and Security; 2015. p. 11–16
77. Davis TA, Hu Y (2011) The University of Florida sparse matrix collection. ACM Trans Mathemat Software (TOMS) 38(1):1–25

Authors and Affiliations

Zuhal Ozturk[1] · Haluk Rahmi Topcuoglu[1] · Mahmut Taylan Kandemir[2]

Zuhal Ozturk
zuhal.ozturk@marmara.edu.tr

Mahmut Taylan Kandemir
mtk2@psu.edu

[1] Computer Engineering Department, Faculty of Engineering, Marmara University, 34854 Istanbul, Turkey

[2] Department of Computer Science and Engineering, The Pennsylvania State University, PA 16802 University Park, USA