Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool

Dong Li
Oak Ridge National Laboratory
lid1@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
Georgia Institute of Technology
vetter@computer.org

Weikuan Yu
Auburn University
wkyu@auburn.edu

***Abstract*—Extreme-scale scientific applications are at a significant risk of being hit by soft errors on supercomputers as the scale of these systems and the component density continues to increase. In order to better understand the specific soft error vulnerabilities in scientific applications, we have built an empirical fault injection and consequence analysis tool - BIFIT - that allows us to evaluate how soft errors impact applications. In particular, BIFIT is designed with capability to inject faults at very specific targets: an arbitrarily-chosen execution point and any specific data structure. We apply BIFIT to three mission-critical scientific applications and investigate the applications vulnerability to soft errors by performing thousands of statistical tests. We, then, classify each applications individual data structures based on their sensitivity to these vulnerabilities, and generalize these classifications across applications. Subsequently, these classifications can be used to apply appropriate resiliency solutions to each data structure within an application. Our study reveals that these scientific applications have a wide range of sensitivities to both the time and the location of a soft error; yet, we are able to identify intrinsic relationships between application vulnerabilities and specific types of data objects. In this regard, BIFIT enables new opportunities for future resiliency research.**

1. Introduction

Resiliency continues to be one of the major cross cutting design goals for high-end computing systems. Today's Petascale systems use a combination of hardware, firmware, and system software techniques to hide many errors from applications, resulting in a mean time between failures or interruptions (MTBF/I) of 6.5-40 hours [1], [2]. Looking forward to Exascale, members of the community expect that both the sheer scale of components, and the move toward heterogeneous architectures, near-threshold computing, and aggressive power management will compound the resiliency challenge so that, with the current techniques, the time to handle system resilience may exceed the mean time to interrupt of top supercomputers before 2015 [3]. To address this compounded challenge, the community is looking toward new techniques where programming models, applications, and perhaps innovative architectural capabilities shoulder more of the responsibility for resiliency than they currently do. For example, programming models can be extended to automatically save critical application state to persistent storage [4], [5]; applications can be modified to use algorithms that remain stable in the face of system errors [6], [7]; and, Exascale architectures can be extended to include nonvolatile memory on each node to provide a fast, persistent store for redundancy or fast checkpointing [8].

One especially difficult problem in resiliency is soft errors: a one-time, unpredictable event that results in bit flips in memory and errors in logic circuit outputs that corrupt and contaminate a computing system's state. Soft errors can result from many causes including packaging material [9], cosmic radiation [10], voltage fluctuation [11], and high temperatures. More importantly, scientific applications are at a pronounced risk of being contaminated by soft errors, because they have long execution times, and they store most of their critical state in the node's DRAM memory, rather than on disk or from an external signal [12]. Because of the high vulnerability of these systems, we must better understand the impact of soft errors on scientific applications.

Yet, the impact of soft errors on a scientific application is understandably complicated. First, soft errors may not be necessarily exposed to the application due to the hardware correction mechanisms (e.g., memory ECC correction). Second, if the soft error propagates through hardware to the application, the corrupted state due to soft errors may not be consumed by an application. For example, the corrupted state may be overwritten by a subsequent write operation. Finally, if contaminated application state persists, then the application may continue to execute correctly, if the corrupted state is not used by the application's future execution. If, however, the application does consume and use the corrupted state, then multiple outcomes are possible: system abort, performance change (degradation), or silently corrupted application data, where the application executes normally both in terms of control flow and performance, but the resulting answer has unacceptable accuracy (and the application has no internal self-consistency checks to detect it). It is also important to note that this last scenario, silent data corruption, can accumulate in the application state and propagate across computation phases [13], [14]. If undetected by explicit, internal application verification, the user will receive erroneous output without a warning to the contrary. If detected, automatic or application level checkpoint-restart can be employed to restart from an

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00

uncontaminated application state.

In order to better understand the specific vulnerabilities to soft errors in scientific applications, we have built an empirical fault injection and consequence analysis tool - named BIFIT for Binary Instrumentation Fault Injection Tool - that allows us to evaluate how soft errors impact applications. Our approach is different than earlier approaches in that we instrument scientific applications by contaminating specific data structures, such as an array of chemical species or grid geometries, and then observe the eventualities of this contamination as the application executes. We, then, classify each application's data structures based on their sensitivity to these perturbations, and generalize these classifications across applications. These classifications can then be used in order to target appropriate resiliency solutions to each data structure within an application. For example, if an application has an in-memory, read-only catalog of materials properties, such as thermal conductivity, then the application could potentially use a resiliency strategy of periodically checksumming the table, and reloading or rebuilding the table if any corruption is found. Furthermore, this strategy might allow this table to be stored in non-ECC protected memory, which may be cheaper and faster than other types of memories. On the other hand, if an array contains a geometry grid for an adaptive unstructured mesh, then the application may employ triple redundancy for the grid and address calculations in order to prevent catastrophic aborts, like segmentation faults, at the expense of using more memory and processing power. Nevertheless, scientists will need strategies to classify and understand the vulnerabilities of their applications at a finer granularity than in the past. In this work, we target each memory object in an application with the aim of potentially identifying and applying a per object resiliency strategy.

A. Contributions

In this paper, we make the following research contributions. (a) We design and implement BIFIT - an empirical fault injection and consequence analysis tool. It provides great flexibility to inject faults into any specified application data structure at any specified execution point. By leveraging application symbols and objects, we bridge the semantic gap between the application and the fault injection site. With the assistance of binary instrumentation, we combine "when" and "where" information to investigate the effects of soft errors. (b) We analyze several important extreme-scale scientific applications in fusion, combustion, and fluid dynamics to reveal the intrinsic relationships among application performance, application state, results, and fault injection points. (c) We develop a general classification scheme that specifically provides scientists with a strategy for understanding which resiliency solution to employ for each data object in their application, ranging from an architectural solution like a hybrid NVRAM-DRAM memory system with different levels of resilience and performance to an algorithm-based adaptive resilience that is effectively resistant to soft errors.

B. Outline

The rest of the paper is organized as follows. §2 explains the fault model used throughout the paper. §3 reviews the related work on fault injection methods and software solutions to soft errors. §4 describes the design of our fault injection tool. §5 then evaluates the susceptibility of three scientific applications to soft errors using error injection experiments. §6 summarizes our observations from the evaluation results, and discusses the implication of our research on future resilience research. Finally, §7 concludes the paper.

2. Fault Model

Traditionally, system errors that influence application reliability and deteriorate a supercomputer's reliability can be broadly divided into two categories: soft errors and hard errors. *Soft errors* are inherently transient in that they are usually caused by temporary environmental factors. By contrast, *hard errors* are either permanent or intermittent (with unstable symptoms at times), and they are usually caused by aged devices or inherent manufacturing defects. Permanent hard errors are easier to detect, because hardware deterioration is often irreversible, and their symptoms tend to be predictable and persistent over time. Hence, they present only a minor threat to application stability in a well-maintained environment [15]. In this paper, we focus on soft errors, because they are problematic to detect, and can exploit vulnerabilities in scientific applications.

The impact of soft errors on applications is complicated. There are at least three outcomes of soft errors:

- *Successful execution*: This outcome indicates two things. First, the application is able to finish without prolonged execution time; second, computational results are either exactly the same as the ones without soft errors, or different but still within acceptable accuracy.
- *Abort*: The system or application aborts, because of segmentation faults, erroneous arithmetic operations (e.g., division by zero) or internal assertion failures in applications.
- *Silent data corruption (SDC)*: The outcome of SDC is more complex as it has several possibilities. SDC occurs when incorrect data is delivered by a computer system to the user (or applciation) without any error being logged. Although the application might be able to finish, computational results may be different from the results without soft errors. In some cases, the result differences can lead to incorrect scientific answers. In other cases, the application can hang for a significant long period of time. For the extreme cases of iterative numerical methods, the application may never converge to be within the expected tolerance for a solution, and, hence, execute forever. The stability of a numerical method can be greatly degraded by the occurrence of SDC errors [14].

Among these outcomes, we examine SDC errors specifically. In general, we classify SDC outcomes into three types: Type 1 extends execution time and results in different solutions; Type 2 only causes different solutions; and, Type

3 only causes prolonged execution time. To detect SDC, many systems rely on redundancy in space or time, which wastes precious system resources and appears unfeasible on an Exascale architecture. In some cases, a computational solution contaminated by SDC may be acceptable if the algorithm's design tests for satisfactory answers within certain thresholds. Examples include many scientific simulations [16] and machine learning algorithms [17], [18] where their algorithm is statistical in nature and can tolerate some variance in the algorithm's answers.

In this paper, we study the impact of soft errors that escape hardware correction and are exposed at the application level. We emulate the impact of soft errors by flipping an arbitrarily chosen bit in the target application's data objects at specific sampled execution points. The data object can be a data segment in global, heap, and stack of the application program (see §4.B for details). We do not inject faults into system libraries (e.g., libgfortran, libgcc and ld-linux-x86-64) and third-party libraries (e.g., netcdf and OpenMPI). Rather, we only inject faults into the scope of data objects which are specific to an application, because we intend to study scientific applications themselves and characterize their vulnerability to silent corruption of data structures. The system libraries and third-party libraries are system-dependent and their impact on applications can vary from one platform to another. We also do not inject faults into the application initialization phase, because applications read input data and setup initial structures (e.g., grid geometries) during this phase. Not surprisingly, this phase is known to be highly sensitive to soft errors and can be easily perturbed by injected faults. Compared to the initialization phase, the vulnerability of the main computational phase is largely unknown; it typically takes most of the execution time, and has a higher possibility to be impacted by SDC. Hence, we inject faults into the main computational phase.

3. Related Work

Soft errors and their impact to application vulnerability and system reliability have attracted a lot of attention over the past several years. Below, we review recent work in related areas.

A. Fault Injection for Soft Error Analysis

Fault injection has been widely adopted as an approach to investigating the impact of soft errors; Table I summarizes a comparison of related work. Bronevetsky and de Supinski [13] implement fault injection at random locations on the stack or heap through manual instrumentation. They specifically target a few iterative solvers such as CG, preconditional Richardson and Chebyshev methods. Although these iterative solvers are widely used in scientific applications, their study is limited to a single computational algorithm and they do not capture the propagation of soft errors across multiple execution phases of a large-scale application. Debardeleben et al. [20] leverages QEMU virtual machine to inject faults at specific assembly instructions. Their method has the potential to profile the vulnerability of applications at the instruction level. However, translating those instructions back to original high-level language instructions on virtual machines is very complex, which makes it difficult to correlate these vulnerability profiles with applications in order to understand the application specific consequences of these faults. Naughton et al. [21] developed a fault injection framework that relies on ptrace or the Linux kernel built-in fault injection framework. Their set of supported injection points target the API-level failures for memory (slab errors and page allocation errors) and disk I/O errors, while our technique focuses on data structures at the application level. Lu and Reed [22] leverage the ptrace system call to halt the target process and overwrite the content of process memory or registers to simulate soft errors. Similar to our work, they categorize the locations of fault injection as global, heap, and stack. However, they randomly insert faults into those memory regions and do not explicitly associate the locations of fault insertion with application data structures. Thus, their work cannot provide a good understanding of the relationship between injected faults and applications. Shantharam et al. [14] and Malkowski et al. [19] manually insert faults into either a random location or a specific data structure. However, their work is limited to specific numerical solvers. It does not investigate a complete application that might have diverse resilience characteristics across execution phases. Sane and Connors [23] use binary instrumentation techniques to inject faults; however, they only instrument random instructions that produce random answers, while we deal with data objects and allow study of specific areas of interest to an application.

B. Software Solutions to Soft Errors

To detect soft errors and mitigate their impact, many software-based reliability techniques have been developed. Huang and Abraham [24] propose algorithm-based fault tolerance. It exploits the algorithmic structure of codes to create efficient, domain-specific fault tolerance schemes. Chen [6] analyzes the block row data partitioning scheme for sparse matrices and derives a sufficient condition for recovering critical data without checkpointing. Ding et al. [25] construct a column/row checksum matrix for matrix multiplication for GPUs. During computation, the partial product matrix is scanned so that soft errors can be detected and corrected at runtime. Du et al. [7] proposes a hybrid fault tolerance solution that combines checkpointing and algorithm-based checksum for dense matrix factorizations. Process-level redundancy [26], [27] has also been investigated as a fault tolerance mechanism. In comparison to checkpoint/restart, this method does not have to roll back execution to a previously known good state for failure recovery. It uses redundant copies of processes to provide fail-over capabilities transparently. However, this method has to be carefully deployed to avoid high costs. Multi-level checkpoint is another promising approach for addressing fault tolerance while avoiding high checkpointing overhead [28], [29], [30]. It allows applications to take frequent, inexpensive checkpoints and less frequent, more resilient checkpoints for better efficiency and reduced load on file systems. Our work provides important application insights for how these types

TABLE I: Comparison of fault injection methods.

Fault injection methods	Application knowledge	User intervention	Fault coverage	Granularity	OS
[13], [14], [19]	Yes	Yes	Specific data objects in iterative solvers	Limited	No
[20]	Very limited	Limited	Specific assembly instructions	No	Yes
[21]	No	No	Slab error, page allocation error and disk I/O error	No	Yes
[22]	No	No	Random addresses in global, heap and stack	Limited	Yes
This paper	Yes	No	All data objects in global, heap and stack	Yes	No

of software solutions might be applied to real applications. For example, our results can be utilized to select application-specific protection points, so that performance and power costs for those software solutions can be further reduced.

4. BIFIT: Fault Injection Tool

We have designed and implemented a binary instrumentation tool, named BIFIT (i.e., Binary Instrumentation-based Fault Injection Tool), based on PIN [31] to inject faults. Instead of exhaustively using manual insertion or heavily relying on the support of operating systems for random fault insertion, BIFIT can insert soft errors into any specified application data object (e.g., scalars, arrays, structures) at any specified execution point.

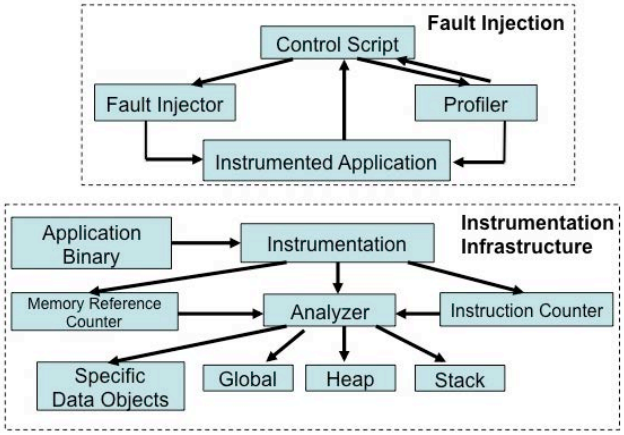

Fig. 1: Diagram of Fault Injection Tool

A. Background: Software-based Fault Injection

Software-based fault injection is the most common technique to investigate the effect of soft errors [13], [14], [19], [20], [21], [22], [28], [15] on real software systems. However, existing software-based fault injection methods have several limitations. First, application knowledge is not fully leveraged to understand the impact of soft errors. Random faults are usually injected to an application state at random points during application execution. Hence, there is a significant semantic gap between application data structures and fault sites. In this case, users have no feedback on which part of the application might need protection to mitigate contamination.

Second, existing fault injection methods do not offer a tight bound between the information of when and where the fault is injected and the impact of soft errors at a fine granularity. It is common for contemporary extreme-scale scientific applications to have hundreds of data objects and have multiple execution phases. For example, we found that the realistic combustion code S3D [16] has 420 data objects and at least four phases to compute different physics such as advection, diffusion, reaction and phase change. This realization indicates that the application may display diverse behaviors and tolerances to injected faults based on the specific data structure and execution point. Hence, it is vital to combine the information of "when" and "where".

Finally, some of current fault injection methods rely on manual fault insertion or heavily involve the support of the operating system. This limitation unnecessarily confines the scope of their fault coverage.

B. BIFIT Overview

In light of these limitations, we have developed BIFIT as shown in Figure 1; BIFIT can insert a fault into any data object in global, heap, and stack memory regions. For global data, a data object can be a global array, a global data structure, or a Fortran common block; for heap data, a data object can be a data segment allocated through the memory allocation subroutines; for stack data, a data object is a stack frame associated with a subroutine invocation.

With this instrumentation infrastructure (see §4.C), BIFIT can enable fault injection at any point during application execution. During the fault injection test for a specific data object, we divide the whole instruction stream into n segments. We then conduct the fault injection test n times, during each of which we flip a randomly chosen bit in the data object right before the first instruction of each segment (see §4.E). This fault injection method emulates soft errors happened at different execution phases of the application. To give the user improved control over the execution point to trigger fault injection, we further define a set of APIs that the users can insert into the application. The APIs use a simple caliper-based approach to trigger, interrupt, or stop the fault injection by replacing calls to the API functions with calls to functions of the tool. This approach allows user to better leverage the application knowledge to analyze the application's vulnerability.

Scientific applications, especially those executed at extreme scales, may have hundreds of data objects. The aforementioned method, although providing for thorough investigations, is a resource-intensive process. It is very important to optimize the infrastructure and instrumentation performance. Therefore, we further developed a profiler that collects information on data objects (see §4.D). Using this profiling information in conjunction with a few control scripts (see §4.F), BIFIT greatly accelerates the testing process. Although BIFIT is used to

study soft errors vulnerability in this paper, it can be used to study the effects of permanent hard errors on global data with only minor modifications.

C. Instrumentation Infrastructure

BIFIT's instrumentation infrastructure provides configurable instrumentation functionality based on the requirements of the profiler and the fault injector. BIFIT instruments the application binary to count the number of executed instructions. To obtain stack information, BIFIT also instruments entry and exit points of all function calls. At the entry point, BIFIT records the base frame address, so that it can maintain a shadow stack of the application. The shadow stack makes it convenient to traverse through the call stack and attribute the effective memory address to the corresponding function's frame, if required by the top-level tool components. The shadow stack also grants the convenience of identifying heap data objects, which will be discussed in §4.E. To obtain heap data information, BIFIT inserts instrumentation at the entry and exit of memory allocation routines (e.g., *malloc()*, *calloc()*, and *realloc()*) to get the heap data object's base address and the object size. BIFIT invalidates the heap data object by inserting an instrumentation point at the entry of *free()*.

To obtain global data information, BIFIT relies on a third party library, *libdwarf*. In particular, given an application binary that has debugging symbols linked into it, BIFIT leverages *libdwarf* to analyze symbol attributes and calculate the size of each global data object. BIFIT then constructs a data structure that encompasses symbol name, base address, and memory size for each global data object. Then, for each fault injection test, the global data information is communicated to the instrumentation infrastructure after the PIN runtime is started and right before the application is executed.

D. Profiler

BIFIT's profiler is used to accelerate instrumentation performance and facilitate the control scripts to identify data objects. It does not impact the application's reaction to the fault injection. Rather, this profiler helps to optimize the testing process. The profiler collects memory access information for all data objects. It also eliminates those global data objects that are never touched by the application. In particular, the profiler detects every memory reference and attributes the memory reference to the corresponding data object, such that we collect total memory references for each data object. Then, during the fault injection test, the memory reference information for the data object specified by the control scripts is input into the fault injector, before the application is executed. During the execution of the instrumented application, the fault injector records the number of memory references to the specified data object and compare it with the input profiling information (i.e., the total number of memory references to this object). Before the application reaches the specified execution point for fault injection, if the fault injector detects that the recorded number of memory references matches the input profiling information (we call it the *p* point), then it will immediately inject a fault, relinquish control of the application, and let the original uninstrumented code run fast forward. We also conclude that the injected fault will limit to the specified data object and never affect other application data. This is based on the fact that the total number of memory references to a data object does not change. After the *p* point is reached, the program will never touch that data object. Hence, any injected fault into that data object will never be propagated to other system states or application data. It does not matter to the application at what execution point we inject the fault after the *p* point. By injecting the fault and detaching instrumentation ahead of the planned execution point for fault injection, we greatly accelerate the instrumentation performance.

The above *p* point-based method has some costs, although it improves overall performance. Particularly, it has to collect the number of memory references to the specified data object before fault injection, which adds extra instrumentation cost. Depending on when the fault is planned to be injected, this cost varies. If the planed injection point is near the completion of the application, then this cost might offset the performance improvement gained from the instrumentation before detaching. Therefore, BIFIT introduces a threshold *N*. If the number of executed instructions at the execution point for fault injection is larger than *N*, then BIFIT does not apply the *p* point-based method. Otherwise, BIFIT does. In our tests in this paper, we conservatively selected *N* as half of the total number of executed instructions of the application. This guarantees performance improvement in our fault injection tests. With the introduction of *N* and the *p* point-based method, we achieve a performance improvement of about *70x* for fault injection tests for 50 data objects.

Of course, the profiler's memory profiling with binary instrumentation is a time-consuming process. In our experiments, we measured about a 200x performance slowdown when running the instrumented application. However, the profiling results can be reused an unlimited number of times in fault injection tests, amortizing the costs of many tests. In addition, to shorten profiling time, we divide the profiler into three tools to profile global, heap, and stack data objects separately. The three tools can run in parallel, which condenses the profiling process. In fact, depending on the user requirement, we can launch any number of instances of the profiler, each of which profiles a specific virtual address range. This brings concurrency into the profiling process and reduces profiling time dramatically.

E. Fault Injector

The fault injector is the core of BIFIT. Its basic functionality is to randomly flip a single bit in the specified data object. In combination with the profiler and under control of the control scripts, the fault injector can inject faults at any execution point with reasonable instrumentation performance.

The fault injector needs to identify the data object. In BIFIT, the Memory (data) Object Identifier (MOI) is consistent across the fault injector, profilier, and control scripts. For global data, this is relatively easy. We construct MOI with the data

object's virtual base address obtained from static analysis on the application binary. At runtime, we can acquire the data object's symbol name and library name based on MOI with the assistant of PIN. This brings the convenience of analyzing fault injection results with application semantic knowledge.

For heap data, we cannot simply rely on static analysis or use symbol names to identify data objects due to dynamic nature of the memory allocation. In addition, frequent memory allocation and deallocation operations within the application make multiple heap memory regions share the same base address and size. These issues make the identification of dynamic data much more difficult. BIFIT's method is to construct an ID by hashing the names of all active subroutines, their associated image names and the data object size when a memory allocation happens. This is feasible by traversing through our shadow stack (see §4.B) and recording the starting address whenever a subroutine is called. The subroutine name and the image name can be obtained from the starting address of the subroutine with PIN. However, in a few cases, we still cannot distinguish the heap data objects with the above method. For those with conflicting hash values, we assign another ID to each of them in an increasing order, following the order they appear during the application execution. To mark whether the heap data object is freed, we also associate a validation flag with each heap data object. If the object is freed before the fault injection point, we will immediately detach the instrumentation and conclude that the heap data object cannot affect the application after the fault injection point. The above identification method positions the dynamic memory allocation within the application context, and differentiates heap data objects. It also contains abundant application information for the convenience of analyzing fault injection results.

For stack data, we use the subroutine name and its associated image name (avoiding subroutine name conflict) to identify the stack data object. In addition, to inject the fault into stack data, we must have explicit address range for each stack data object. However, the call stack dynamically grows and shrinks at runtime. Therefore, we monitor the stack pointer for the topmost active routine in the stack, so that we ensure the fault will be injected into the right data object.

In general, the above memory identification method is independent of architecture and compiler. The IDs remain constant throughout profiling and fault injection phases.

F. Control Script

The BIFIT control script is the only component that the user needs to directly use. It integrates all components of the tool and schedules fault injection tests as configured by the user. The basic functionality of the script is threefold: test configuration, performance measurement, and result collection.

For test configuration, the user can specify which data objects into which they want to inject faults. They can be either very general (e.g., global data, heap data, or stack data), or very specific data objects (e.g., specific global array, or specific subroutine stack); and, the user can also specify at which execution points to inject a fault. The fault can happen either multiple times, or just once as done in our tests. With the support of our instrumentation infrastructure, the user has great flexibility to design a significant number of fault injection scenarios.

The control script also measures the application performance for each fault injection test. Because of binary instrumentation cost, the application execution time is extended even without the fault. This situation makes the task of quantification of soft errors impact on the application execution time more complex.

BIFIT uses the following method to filter out instrumentation cost. For each specified memory object and specified execution point, we conduct two tests, one with fault injection, and the other without fault injection but with instrumentation enabled. We measure execution times for both tests (t_1 and t_2 respectively). We also measure the execution time with neither instrumentation nor fault injection (t_0). Then the performance impact due to the soft error (P) is calculated as:

$$P = (t_1 - t_2)/t_0 \quad (1)$$

Some fault injection tests cannot be performed. For example, before the fault injection point, a heap memory object may be already freed, or a subroutine associated with a stack memory object may be already returned. For those cases, the above no-fault-injection instrumentation test is skipped. The control script directly calculates P as 0, which represents the application execution time is not affected.

The control script records application responses throughout the fault injection testing. The application reactions can be collected from both application-specific status and system outputs. The application-specific status includes checkpoint files, history logs, result figures, etc. They must be customized according to application algorithms and user requirements. The system outputs include the outputs from the operating system kernel's message buffer, the standard error stream, and application exit status. In addition, the control script will kill the application if the application "hangs." BIFIT deems that an application in this state if its execution time grows beyond a threshold. The threshold is determined by the user and preset at the configuration phase. In our tests, we set it as 10 (i.e., the execution time is extended 10 times longer). This is in consistent with previous fault injection research [13].

5. Application Results

We apply BIFIT to three extreme-scale scientific applications to assess their soft error vulnerability. The applications characteristics are summarized in Table II. This section presents the results of our investigation. We use PIN 2.10 and gcc 4.4.6 on Linux 2.6.32 in our evaluation. We apply fault injection tests at three different execution points for each data object (i.e., $n = 3$). For a data object in global or heap, the fault injection site (i.e., a bit in the data object) remains the same across the three fault injection tests to ensure fair comparison. We instrument only one MPI task for each application, but BIFIT has the ability to instrument all tasks concurrently.

TABLE II: Applications.

Application	Input problem	Global Objs	Heap Objs	Stack Objs	Tests
Nek5000	3D vortex problem with default problem size	456	151	617	3672
S3D	Grid dimension: 15x15x15, default setting	128	172	120	1260
GTC (v2.0)	Poloidal grid points=192, micell=3, mecell=5	207	1136	59	4206

We evaluate the effects of injected faults on applications from the aspects of both application run time and application states/outputs. To investigate the effects on application run time, we categorize those executions with extended run times into 7 classes, based on how long the run time is extended. Specifically, the performance loss falls into one of the following categories: (0.05, 0.1], (0.1, 0.2], (0.2, 0.4], (0.4, 0.6], (0.6, 0.8], (0.8, 1] and >1). We regard an execution with performance loss no larger than 5% as normal performance to tolerate regular performance variance. We count the number of data objects that fall into each class at each fault injection test in Figures 3, 5, and 7. Assessing the effects of soft errors on application states/outputs is challenging. For some scientific applications, a deviation from the no-fault application states/outputs may still be valid. Fully understanding the validness of states/outputs demands very detailed domain-specific knowledge and also reflect subjective opinions of the users. In our tests, we check checkpoint files of the applications in the last time step. These checkpoint files hold application-critical computation states. We check them to determine whether they are different from those of the executions without fault injection. We count the number of data objects that have an injected fault that leads to the differences in checkpoint files and critical application outputs in Table III. In doing so, we enforce an objective judgment on the effects of soft errors on application states/outputs, and do not judge the validness of the applications states/outputs.

Gx, Hx and Sx (x=1, 2, and 3) in the following figures and tables represent the fault injection tests performed at the x execution point and at the global, heap, and stack data objects respectively. To make comparison easier across the 3 execution points and across different type of data objects, we report the percentage of data objects within each type (i.e., global, heap or stack). For volatile heap and stack data objects, if they are not available at the specified execution point, the impact of fault injection on the application is none. They are not displayed in our results because of their trivial impacts on applications, but this kind of fault injection tests is counted during the testing campaigns.

A. Nek5000

Nek5000 [32] is a dynamic solver simulating unsteady incompressible fluid flow with thermal and passive scalar transport on two- and three-dimensional domains. It is widely used in a broad range of scientific research, including thermal hydraulics of reactor cores, transition in vascular flows, ocean current modeling and combustion. Nek5000 has been deployed on the Jaguar supercomputer at Oak Ridge National Lab. It is a potential application to be deployed in future Exascale systems.

TABLE III: The number of data objects (percentage) in applications where the fault injection results in the differences of the checkpoint file and outputs.

	G1	G2	G3	H1	H2	H3	S1	S2	S3
Nek5000									
checkpoint file	18	18	18	6.8	7.5	6.8	3.5	1.9	1.8
critical points	16	18	18	6.8	7.5	6.8	11	5.0	3.1
S3D									
checkpoint file	3.1	3.9	25	3.1	3.2	3.2	0	0	0
C11-Y	1.6	0.78	0	0	0	54	0	3.3	0
C12-H2	0	0	0	0	0	54	0	3.3	0
C17-H2O	7.8	7.8	0	0	0	54	0	3.3	0
C22-CO	1.6	0.78	0	0.63	0	54	0	3.3	0
GTC									
checkpoint file	29	34	60	0	0	0.7	0	0	0
history file	28	33	61	1.1	1.1	2.8	0	0	5.2
data1d file	28	33	60	1.1	1.1	2.1	0	0	5.2

Figure 2 summarizes the fault injection results. We notice that the application success rate with faults injected in global data is higher than that in other types of data objects, statistically demonstrating the insensitivity of global data to those injected faults. Type 2 SDC errors (i.e., the error only resulting in different states/outputs) are more common than the other types of SDC errors, which means that the execution time is less impacted than the application states/outputs. We also notice that faults injected in stack data at the second fault injection time point tend to impact applications more often than other cases (e.g., having more Type 3 SDC errors), while we do not find the similar error rates at the first and the third fault injection time points. This shows that the application is sensitive to when the fault is injected.

Figure 3 displays the impact of soft errors on the execution time. We first notice that the execution times are widely distributed between the 7 classes. The faults injected in the global and heap data objects tend to result in longer execution time. Particularly, the percentage of data objects in the class ">1" is higher. Also, the same data object with faults injected at different time points can result in different execution times. This is especially demonstrated by the significant difference of the execution time distribution in the heap data objects at the time points 1 and 3.

Table III presents the impact of soft errors on the application state (i.e., the checkpoint file) and the outputs (i.e., 3 velocity components and 1 pressure component at the final time steps for 6 critical points). We notice that the faults injected in the global data objects tend to impact application states/outputs more than those in other kinds of data objects. This observation is consistent throughout the 3 fault injection time points.

B. S3D

S3D [16] is a massively parallel direct numerical solver (DNS) for the full compressible Navier-Stokes, total energy, species and mass continuity equations coupled with detailed

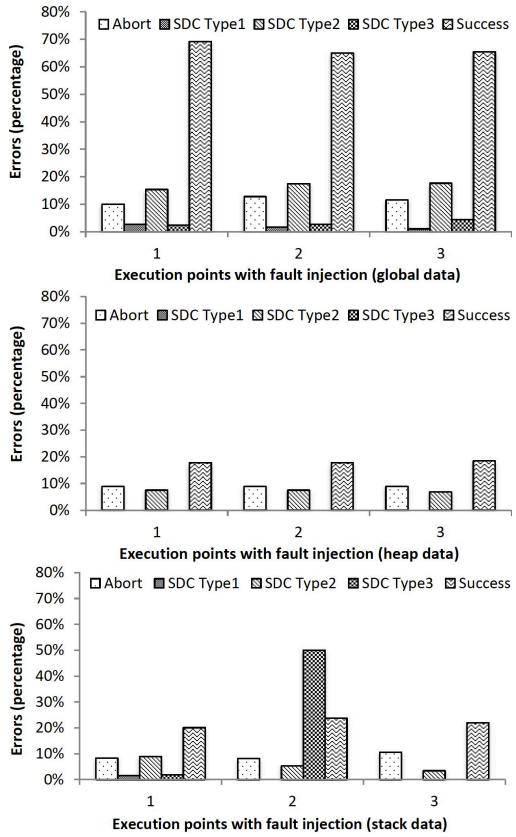

Fig. 2: Summary of fault injection results (Nek5000)

chemistry. S3D can greatly advance our basic understanding of turbulent combustion processes and thus improve efficiency of combustion devices. S3D has been deployed in Jaguar supercomputer and is planned to be deployed in the future Titan supercomputer in Oak Ridge National Lab.

Figure 4 summarizes the fault injection results. In S3D, the global data objects with fault injected are responsible for most of the abort errors throughout the application execution. S3D seems to be more sensitive to the fault injection than Nek5000. In particular, except the heap data objects with faults injected at the first time points, the "success" rate in other data objects and time points is about 10%-20% smaller than that of Nek5K. Also, Type 3 SDC errors, instead of Type 2 errors in Nek5K, are more common than other SDC errors.

Figure 5 displays the impact of soft errors on the execution time. The faults injected in the global data objects have small impact on the execution time; the heap objects with fault injected at the time points 2 and 3 greatly extend the execution

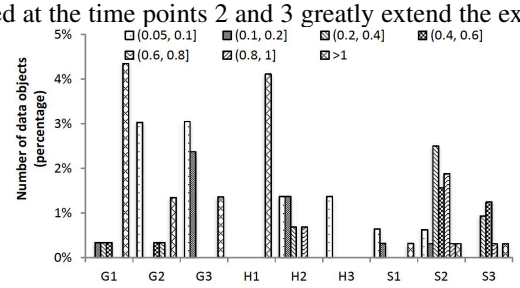

Fig. 3: The distribution of extended run time (Nek5000)

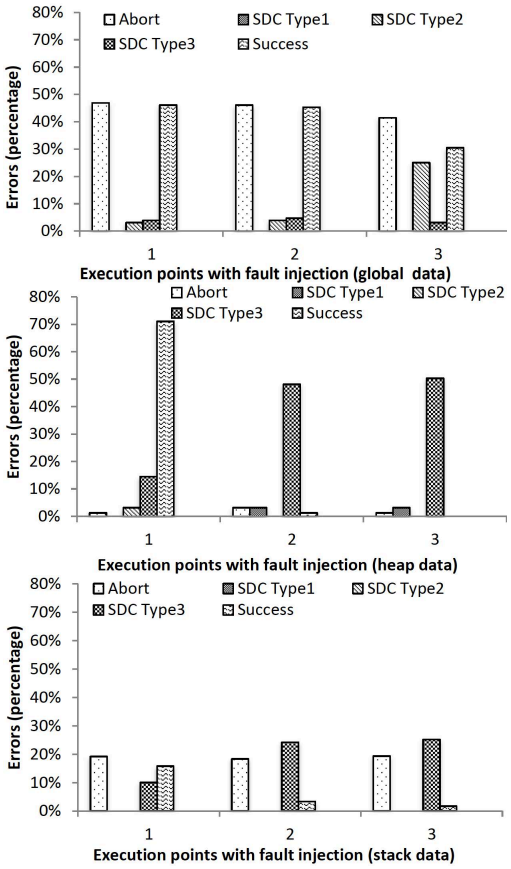

Fig. 4: Summary of fault injection results (S3D)

time. Most of the cases with application hangs also happen in the heap objects. The stack objects with faults injected have diverse impact on the execution time, depending on when the fault is injected.

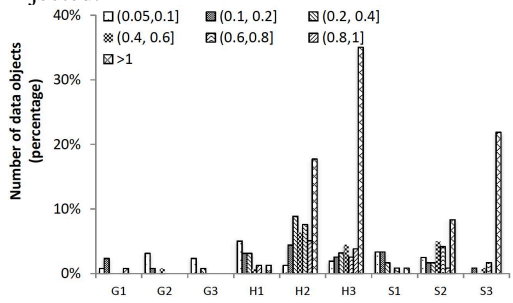

Fig. 5: The distribution of extended run time (S3D)

Table III presents the impact of soft errors on the application state (i.e., the checkpoint file) and the outputs (i.e., 4 parameter values at the final time). S3D has more than 64 output parameters. We only show four of them as a representative set. The other parameters are either not affected by fault injection or have the similar reactions as the four. Similar to Nek5000, we observe that the faults injected in the global data objects tend to impact application states. This is especially true at the third fault injection time point. The faults injected in heap data objects tend to affect the output parameters while do not impact the application states as much as those in global data objects do. In addition, the fault injection at a late execution phase (i.e., the execution point 3) generally affects

the application more easily than that at other execution phases.

C. GTC

GTC [33] is a massively parallel, particle-in-cell code for turbulence simulation in support of the burning plasma experiment, the crucial next step in the quest for next generation (fusion) energy. GTC has been deployed in Jaguar supercomputer in Oak Ridge National Lab.

Figure 6 summarizes the fault injection results. We first notice that Type 1 SDC errors in the fault injection results of global data objects, which are not often observed in the other two applications. We also notice the high abort rates in the fault injection tests for stack data objects. In addition, unlike Nek5000 and S3D that typically have unbalanced Type 2 and Type 3 SDC errors, such errors in GTC have similar possibility to appear in all data objects.

Figure 7 displays the impact of soft errors on the execution time. Unlike Nek5000 and S3D, the execution time is significantly extended: most cases have either more than doubled execution times or simply hang. This indicates that some data objects in GTC, once corrupted, lead to slow converge to the expected accuracy range and these data objects more commonly exist in GTC than in the other applications.

Table III presents the impact of soft errors on the application state (i.e., the checkpoint file and the history file) and the outputs (i.e., the data1d file). It is clear that the injected faults in the global data objects have much higher possibility to impact application states and outputs than those in other data objects. This is especially true at the third fault injection time point.

6. Observations

The results in §5 statistically reveal several facts. First, the application is sensitive to when and where the fault is injected. Depending on when the fault is injected, the same affected data object can have different impact on the application. Second, the global data objects are usually tightly related to critical application outputs and states. They should be well protected to guarantee result correctness. Third, applications are different from each other in vulnerability. They tend to react differently to different soft errors, depending on their inherent algorithms and workload characteristics. This indicates that a domain specific resilience design might be able to better detect soft errors or tolerate faults. Fourth, the soft errors happened at the late execution phases tend to have larger impact on the application states and outputs than those happened earlier in the execution phases. This indicates the possibility of using adaptive resilience designs to balance resilience costs and application reliability. Fifth, the application execution time is very sensitive to soft errors. It is common to double (or even more) execution time with soft errors. Hence, software-level symptom based fault detection techniques may monitor execution time for anomalous software behaviors.

A. Application Implication

We link our observations with applications and use application knowledge to further understand the results. We look

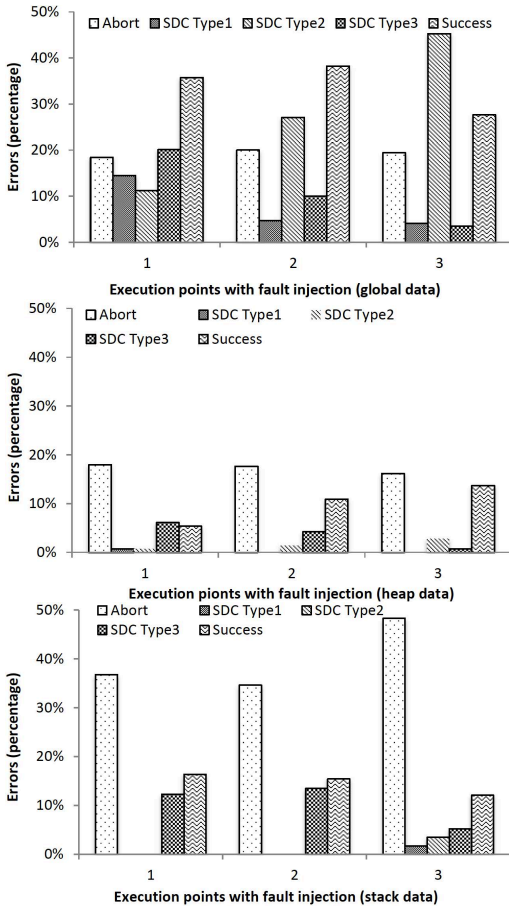

Fig. 6: Summary of fault injection results (GTC)

closely at those data objects in the global and heap of Nek5000 whose corruptions cause applications to abort. We find that 78% of them in global and 100% of them in heap are small data objects (smaller than 1K). For global data, these small data objects represent critical computation parameters, for example, the ones recording size and identity of domains, the index array for accessing multi-grids and the handle used in calling the coarse-grid solver; for heap data, these data objects are used for computing-dependent intermediate parameters and results, for example, the communication tree formed during runtime.

We also examine those data objects in the global and heap of Nek5000 whose corruptions lead to differences in the checkpoint file (a binary file). We compare the checkpoint

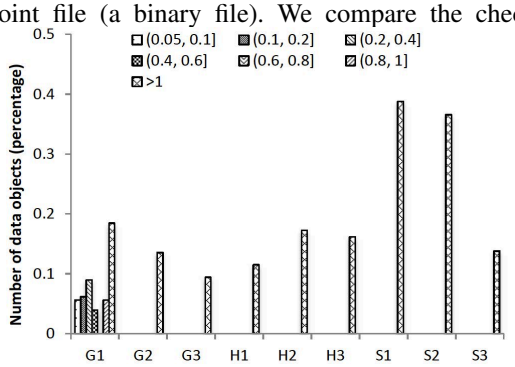

Fig. 7: The distribution of extended run tim (GTC)

files between the faulty one and the regular one, and report how different they are in terms of percentage of different bytes in the file. We then correlate file difference, data object size and application knowledge. Some results are summarized in Table IV. In these results, if a data object with fault injection causes difference in the checkpoint file in multiple execution time points, we count them multiple times correspondingly to fully understand the effects of fault injection on this data object. From the results, we notice that most data objects that cause significant difference (>75%) in the application states are small. After linking the observation with application knowledge, we find that they are also critical computation parameters, such as the extrapolation terms for magnetic field and perturbation parameters. This result is consistent with the above examination on data objects that cause application abort.

TABLE IV: The number of data objects in the global and heap of Nek5000 that lead to differences in the application checkpoint file.

File Difference	0-25%	25%-50%	50%-75%	75%-100%
obj size <=1KB	72	5	14	29
obj size 1KB-1MB	60	1	1	7
obj size >1MB	0	0	0	0

We have the similar observations in S3D and GTC. For example, in S3D, we find look-up tables that contain coefficients for linear interpolation and global arrays for chemistry properties; in GTC, we find quite a lot of control parameters and auxiliary radial interpolation arrays. These data objects are typically small and the corruptions in them lead to either abort or quite different application states. These observations demonstrate the necessity of classifying soft error vulnerabilities in data objects and use diverse resilience policy to selectively protect data.

We further notice that the velocity field (a data object in Nek5000), if involved in a stability calculation, is highly sensitive to soft errors. Once it is corrupted, the application can abort. However, if it is involved in the turbulence computation, the soft error in the velocity field can be averaged out, so that the computation results are still correct, although the execution time is extended. This is a realistic example of how the application's vulnerability to a data structure varies across the execution phases.

B. New Research Opportunities

BIFIT and our fault injection results provide opportunities for two additional directions of research: hybrid memory systems and algorithm-based resilience.

Hybrid memory systems: The hybrid non-volatile memory-DRAM/SRAM systems have been studied to leverage non-volatile memory for either performance improvement or power saving [34], [35], [36]. However, it has not been studied for resilience purpose. Non-volatile memory (NVM) is resistant to the particle strikes on the transistor and much less impacted by thermal fluctuation than DRAM/SRAM [37]. Therefore, it can be leveraged to place those data objects whose fault affection can seriously impact application performance and output accuracy. By placing those data objects into NVM, we may simplify those costly hardware checksum and ECC mechanisms to improve performance and save power.

Algorithm-based resilience: The algorithm-based resilience leverages algorithmic structures of codes to create domain-specific fault tolerance scheme (see §3.B for details). Since BIFIT has the flexibility of fault injection at any specified data structures at any execution phase, it provides a powerful tool to optimize algorithm-based fault tolerance schemes. For example, if a specific critical data is not impacted by other faulty data structures at certain execution phases, then the algorithm based checksum can be simplified, which will result in performance improvement.

7. Conclusions

Resilience is becoming an important concern in the future Exascale systems. Soft errors will significantly impact system resilience. Understanding the impact of soft errors on scientific applications is critical for future Exascale system designs. In this paper, we propose a fault injection tool, named BIFIT, based on binary instrumentation to simulate single bit errors. BIFIT enables fault injection at any specified application data object in global, heap and stack at any specified execution point. With the assistance of a profiling tool and control scripts, we greatly improve the instrumentation performance of BIFIT, and fully automate the fault injection deployment and result collection. We apply BIFIT to three realistic extreme scale scientific applications, and perform thousands of injections to simulate single bit flip errors. We comprehensively investigate the impact of soft errors on application execution time, states and outputs. Our results statistically indicate the relationship between the application reaction, and the fault injection sites and times. BIFIT and our results provide insightful information for the future resilience research.

Acknowledgments

The paper has been authored, in part, by Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract #DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. This research is sponsored in part by an NSF award CNS-1059376, and by the Office of Advanced Scientific Computing Research in the U.S. Department of Energy.

References

[1] C. Wang, F. Mueller, C. Engelmann, and S. Scott, "Hybrid Checkpointing for MPI Jobs in HPC Environments," in *16th International Conference on Parallel and Distributed Systems*, 2010.

[2] K. Kharbas *et al.*, "Combining Partial Redundancy and Checkpointing for HPC," in *International Conference on Distributed Computing Systems*, 2012.

[3] F. Cappello *et al.*, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[4] IBM and the compute grid programming models, "POJO Programming Model," http://pic.dhe.ibm.com/infocenter/wxdinfo/v6r1.

[5] C. Makassikis, V. Galtier, and S. Vialle, "A Skeletal-Based Approach for the Development of Fault-Tolerant SPMD Applications," in *International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2010.

[6] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, 2011.

[7] P. Du *et al.*, "Algorithm-based fault tolerance for dense matrix factorizations," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012.

[8] X. Dong *et al.*, "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[9] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, 2005.

[10] T. Karnik *et al.*, "Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18 μ," in *2001 Symposium on VLSI Circuits Digest of Technical Papers*, 2001, pp. 61–62.

[11] S. Krishnamohan and N. Mahapatra, "A Highly-Efficient Technique for Reducing Soft Errors in Static CMOS Circuits," in *Proceedings of the IEEE International Conference on Computer Design*, 2004.

[12] M. Heroux, P. Raghavan, and H. Simon, *Parallel processing for scientific computing (Software, Environment and Tools)*. Society for Industrial Mathematics, 2006, no. 20.

[13] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008.

[14] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 152–161.

[15] X. Li, M. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, 2010.

[16] E. Hawkes, R. Sankaran, J. Sutherland, and J. Chen, "Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models," in *Journal of Physics: Conference Series*, vol. 16, 2005, p. 65.

[17] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *IEEE International Symposium on Parallel and Distributed Processing*, 2009.

[18] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna, "Exploiting the forgiving nature of applications for scalable parallel execution," in *IEEE International Symposium on Parallel and Distributed Processing*, 2010.

[19] K. Malkowski, P. Raghavan, and M. Kandemir, "Analyzing the soft error resilience of linear solvers on multicore multiprocessors," in *IEEE International Symposium on Parallel and Distributed Processing*, 2010.

[20] N. DeBardeleben *et al.*, "Experimental Framework for Injecting Logic Errors in a Virtual Machine to Profile Applications for Soft Error Resilience," in *Workshop on Resilience in High Performance Computing in Clusters, Clouds and Grids*, 2011.

[21] T. Naughton *et al.*, "Fault injection framework for system resilience evaluation: fake faults for finding future failures," in *Proceedings of the 2009 workshop on Resiliency in high performance computing*, 2009.

[22] C. Lu and D. Reed, "Assessing fault sensitivity in MPI applications," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[23] H. Sane and D. Connors, "A framework for efficiently analyzing architecture-level fault tolerance behavior in applications," University of Colorado Denver, Department of Electrical Engineering, Tech. Rep., 2008.

[24] K. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 518–528, 1984.

[25] C. Ding *et al.*, "Matrix multiplication on gpus with on-line fault tolerance," in *9th International Symposium on Parallel and Distributed Processing with Applications*, 2011.

[26] A. Shye *et al.*, "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," in *International Conference on Dependable Systems and Networks*, 1976.

[27] K. Ferreira *et al.*, "Evaluating the viability of process replication reliability for exascale systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[28] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[29] N. Vaidya, *A Case for Multi-Level Distributed Recovery Schemes*. Texas A&M University Technical Report, 1994.

[30] E. Gelenbe, "A model of roll-back recovery with multiple checkpoints," in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 251–255.

[31] V. Reddi, A. Settle, D. Connors, and R. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 workshop on Computer architecture education*, 2004.

[32] P. Fischer and J. Lottes, *nek5000 Web page*. Web page: http://nek5000.mcs.anl.gov, 2008.

[33] Z. Lin, "Tuning CUDA Applications for Fermi," http://phonenix.ps.uci.edu/GTC.

[34] L. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *International Conference on Supercomputing*, 2011.

[35] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecturing Phase Change Mmemorhy as a Scalable DRAM Architecture," in *International Symposium on Computer Architecture*, 2009.

[36] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *ACM/IEEE Design Automation Conference*, 2006.

[37] G. Sun, E. Kursun, J. River, and Y. Xie, "Exploring the vulnerability of CMPs to soft errors with 3D stacked non-volatile memory," in *International Conference on Computer Design*, 2011.