Exploring the Effect of Compiler Optimizations on the Reliability of HPC Applications

Rizwan A. Ashraf
Oak Ridge National Laboratory
Oak Ridge, TN 37831
Email: ashrafra@ornl.gov

Roberto Gioiosa and Gokcen Kestor
Pacific Northwest National Laboratory
Richland, WA, 99352
Email: {roberto.gioiosa, gokcen.kestor}@pnnl.gov

Ronald F. DeMara
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL 32816
Email: demara@ucf.edu

***Abstract*—The strict power efficiency constraints required to achieve exascale systems will dramatically increase the number of detected and undetected transient errors in future high performance computing (HPC) systems. Among the various factors that effect system resiliency, the impact of compiler optimizations on the vulnerability of scientific applications executed on HPC systems has not been widely explored. In this work, we analyze whether and how most common compiler optimizations impact the vulnerability of several mission-critical applications, what are the trade-offs between performance and vulnerability and the causal relations between compiler optimization and application vulnerability. We show that highly-optimized code is generally more vulnerable than unoptimized code. We also show that, while increasing optimization level can drastically improve application performance as expected. However, certain cases of optimization may provide only marginal benefits, but considerably increase application vulnerability.**

I. Introduction

Achieving exascale performance within constrained power budgets [13] will require the deployment of new hardware technologies, such as massively multi-threaded cores that operate at near-threshold voltage (NTV) [10] while simultaneously reaping the benefits of shrinking process technology. This strict power budget will also limit the amount of circuit hardening, guard-bands, ECC, and other resilience mechanisms that can be incorporated to detect and/or correct errors at the hardware level [17]. This will exacerbate the number of undetected errors, including silent data corruption (SDC), which may significantly corrupt application correctness, resulting in wasted energy and resources. The use of these new technologies, together with the sheer number of components in an exascale-class high-performance computing (HPC) system, will cause the number of soft and hard errors to increase dramatically [32], [3]. Without novel resilience solutions, the mean time to failure (MTTF) of an exascale supercomputer will be considerably lower than current petascale systems, to the point that it will become difficult to correctly complete a parallel application [13].

Many factors affect the resilience of a HPC system, including the application's algorithm and characteristics, the scientific libraries used, the programming language used, the environmental conditions of the system (e.g., altitude), and the aging of the hardware components [34], [40]. Among these factors, compiler optimizations have been usually considered of secondary order and their impact on resiliency has not been widely investigated, especially in the context of HPC systems. Such studies are critical for HPC applications since compiler optimizations are one of the most successful ways to improve performance with relatively low effort from the programmer. Modern compilers perform very sophisticated code analysis and code transformation almost transparently to the user. The most common compiler optimizations include loop unrolling, memory pre-fetching, instruction reordering to hide memory latency and increase instruction-level parallelism, and elimination of dead branches and unused global variables. Moreover, compilers can exploit hardware-specific optimizations, such as special instructions and vector units, transparently to the user. The impact of these code transformations and optimizations on performance is substantial to the point that compiler optimizations are considered essential to achieve high performance and efficiency.

Compiler optimizations and/or software/algorithmic modifications, however, also impact the vulnerability of the application, which can affect the overall resilience of the computing system [4], [20], [31]. For example, code optimization can increase the utilization and the throughput of out-of-order processors, which generally improve performance. On the other hand, assuming that faults occur randomly in the processor, higher utilization implies higher sensitivity to faults, as there are more processor's components that operate on application's instructions at the same time. Similarly, increasing data locality may reduce the number of cache misses, thus errors occurring in a cache line will have a lower probability to propagate to main memory. The same optimization, however, may increase the time a certain value resides in a register

before being refreshed. Thus, an error in the processor register will stay in the processor longer and have a high probability of propagating into the application's data structures and incur disruptive effects.

In this work, we investigate the interplay between compiler optimizations and the vulnerability of HPC parallel applications in the presence of radiation-induced soft errors which effect Static Random-Access Memory (SRAM) elements as elaborated in Section II. This fault model has been employed in various other studies, e.g., LLFI [39], KULFI [30], VULFI [29]. As a result of this work, we analyze the HPC-specific trade-offs between performance improvement and resilience, and the implications of scaling parallel applications beyond a single multi-core compute node. Specifically, we seek the answer to several important questions, such as:

- *"Is performance gain obtained by a given optimization worth the increase in vulnerability?"*,
- *"Are there optimizations that provide the same level of performance, but decrease vulnerability?"*,
- *"Do memory operations modified by optimization levels have a significant impact on vulnerability?"*

Previous work has focused on investigating the effects of compiler optimizations on the vulnerability of multi-programmed workloads [9], [19] or fault injection studies in the embedded system domain [25], [22]. However, as HPC applications and systems are more vulnerable to soft errors due to their distributed nature and scale, we undertake this task to explore opportunities for design of exascale systems. In this work, we analyze several mission-critical and proxy applications from the DOE domain on a cluster of dual-socket AMD Opteron 6227 (Interlagos) compute nodes with up to 512 cores. Our results show that optimized code performs better than unoptimized code but also that optimized code is more vulnerable to failures. Moreover, we show that both performance and vulnerability generally follow the same trend and increase with increasing level of code optimization. We also show that, in some cases, additional compiler optimizations do not yield additional performance, but significantly increase application vulnerability. In other cases, a small performance penalty caused by not using some compiler optimization may decrease the vulnerability of the application. Finally, we analyze the causal relationships between compiler optimizations and application vulnerability and the main reasons that induce application crashes.

The remainder of this work is organized as follows: Section II describes our methodology; Section III presents our hardware and software setup; Section IV shows our experimental results; Section V presents the related work; Section VI presents the conclusions of this work.

II. Methodology

In this section, we describe the fault model used in this work, our methodology for accelerated fault injection, and the compiler optimizations considered.

A. Fault Model

Faults occur at hardware level as a result of physical phenomena or exposure to alpha particles. Generally, faults are categorized in two main categories: *hard faults*, which are either permanent or intermittent and are typically the result of transistor aging effects or malfunctioning devices; *soft faults*, which are transient in nature and typically caused by environmental conditions, such as radiation effects [7], that manifest in the form of bit flips at the circuit-level. With the development of technologies needed for low-power operation [10], technology scaling of electronic components and the need for higher temperature tolerance and the scale (number of components) required to achieve exascale efficiency [13], soft faults are expected to become predominant [16], [32].

In this work, we analyze the intrinsic vulnerability of HPC parallel applications to errors independently of the underlying hardware, i.e., we analyze the impact of transient errors on parallel applications once a fault is articulated at the application level. Our methodology follows the general idea proposed by analysis such as the *program vulnerability factor (PVF)* [31], where a program is analyzed in terms of its instruction flow and the probability that a fault occurring at register-level contaminates the application data structures. Given that HPC systems are built out of commodity hardware, we refer to previous work to assess the probability that a fault at circuit-level induces a bit flip at the register-level [4], [7].

In order to emulate the occurrence of transient hardware faults, we use accelerated error injection techniques [14], [39], [40], [26] and analyze the application outcomes. In particular, we randomly inject single-bit flips [7], [6], [36], [1] during the execution of an application by leveraging the LLVM compiler framework and extending an existing fault injection tool, LLFI [39].

Transient errors may occur any time during the execution of an application and can result in different outcomes. Although many possible classifications are possible, in this work we extend the classification proposed in [6], [7], [4], [36], [37] to account for the specific characteristics of HPC applications. We classify the possible application outcomes resulting from fault injection into the following classes:

Masked: A fault is masked if it does not effect the final output of the application nor the total execution time. Note that this does not necessarily mean that no data structure is affected by the fault, but rather the application is able to tolerate the fault and still produce correct outputs [1]. Moreover, the result of an HPC application is usually considered acceptable if it falls within certain error margins, thus small variations in the final result may be tolerated by the algorithm.

SDC: If the final result computed by the application is outside of the allowed error margins, the solution is not acceptable, thus the output is considered wrong.

Prolonged execution: Some applications may be able to tolerate transient faults by performing extra work to refine the current solution. These applications provide

some form of inherent fault tolerance in their algorithms, though at the cost of delaying the output.

Crash: Faults which induce program crashes or hangs.

Several definitions of application vulnerability have been proposed in the literature [31], [9], [18], [40], some of which require a complete understanding or estimation of the utilization of internal processor functional units and storage. However, collecting these metrics on a cluster of distributed compute nodes is computationally expensive. Moreover, most of these metrics are scalar values [31], [18], [40] that do not take into account the effects of message interleaving common in MPI applications, or are biased towards the application's execution time, i.e., shorter applications appear more reliable [9]. Given the complexity and the scale of the applications tested in this work, we use a practical approach based on the observation of the application characteristics and how these change, when applying different compiler optimizations. We are interested to understand the performance and vulnerability trade-offs and the interplay between compiler optimizations and the application sensitivity to faults. This approach is orthogonal to the vulnerability metrics already proposed and can be used in conjunction with them.

B. Fault Injection Module

Several fault injection tools have been proposed in the literature [30], [39]. In this study, we use LLFI [39], an LLVM based fault injection tool that instruments sequential applications at the level of LLVM intermediate representation (IR) with function calls into the runtime fault injector. At runtime, a fault is injected into one of the registers of a randomly selected instruction in the form of a bit flip. We extended LLFI to account for the distributed and parallel nature of HPC applications. We inject a single-bit fault into one of the application's MPI processes, randomly selected in each experiment. However, the fault can propagate to other MPI processes, corrupting their memory address spaces [1].

It should be pointed out that there are limitations of using compiler-based fault-injection tools. For example, faults are injectable only into live user-programmable registers, and dormant and/or OS-visible registers. This is a common problem for software-implemented fault injection tools.On the other end, simulated environments allow to insert faults into dead or non-programmable registers, but it would drastically limit the scope and scale of this study.

C. Compiler Optimizations

In this work, we use the `clang` 3.4 compiler. Although modern compilers provide the users with specific options to optimize their code, typically set of optimizations are grouped into higher level options `O0`, `O1`, `O2`, `O3`. Table I lists the `clang` optimizations applied at each optimization level incrementally from `O0`, to `O3`. Several optimizations are applied at level `O1`, including loop unrolling, rotation and simplifications, function inlining, memory dependency analysis, etc. At level `O2` the compiler performs store and loop vectorization, removes unreachable global variables, and eliminates redundant instructions. At optimization level `O3`, the compiler promotes arguments passed by reference to arguments passed by value. The most important compiler transformations which may effect the memory operations in program code include:

- *Loop Vectorization:* transform data dependence graphs that do not exhibit cycles between iterations into single instructions on multiple data items consisting of a range of array indexes;
- *Memory to Register:* promotes memory references to register references to increase register utilization;
- *Dead Load/Store Elimination:* redundant stores are eliminated; global values are enumerated to enable elimination of redundant loads;
- *Register Renaming:* reassignment of variables to remove output dependencies or optimize register use.

Most of the above transformations are applied as reported in Table I. For instance, `-gvn` global value numbering is used to eliminate redundant load instructions during `O2`. Similarly, `-loop-vectorize` is performed at the same optimization level. We also consider the `-mem2reg` optimization, which promotes memory references to register references, resulting in reduced number of memory references and increased register pressure. We remark that we only considered *safe* optimizations that do not induce code misbehavior or incorrect application output, thus eventual errors are caused by the injected faults.

III. Experimental Setup

Next, the hardware/software setup and applications used in this work are described.

Hardware and Software Platform: We performed our experiments on a cluster of dual-socket AMD Opteron 6227 (Interlagos) compute nodes. Each processor socket is comprised with 32 cores running at 2.2 GHz. Cores feature a 64 KB L1 data cache while pair of cores share a 64 KB L1 instruction cache and a 256 KB integrated L2 cache. Compute nodes are equipped with 64 GB of DRAM divided into 4 NUMA domains. Each NUMA domain consists of 8 processor cores that share a memory controller and a 8 MB L3 cache. The compute nodes are interconnected through an Infiniband communication network. The system runs Linux 3.9.0 while all applications are compiled with `LLVM/clang` version 3.4 and `OpenMPI` 1.7.4. The compiler optimizations are applied prior to the fault instrumentation performed by LLFI, such that error injection code is not effected by the optimizations. In addition, error injection is not performed in external library calls such as MPI functions, hence we focus on the fault characterization of the application itself.

Applications: We analyze the impact of compiler optimizations on several important DOE applications and benchmarks taken from various suites. In particular, in this study, we analyze *LULESH2*, *LAMMPS* and *MCB* from the CORAL

Tab. I: Optimizations applied at each level by `clang`. Each column shows the additional optimizations added to the previous level (optimizations between parenthesis are removed).

Opt-level	Optimizations Applied (in order)
`O0`	-targetlibinfo -datalayout -notti -basictti -x86tti -preverify -domtree -verify
`O1`	-no-aa -tbaa -basicaa -globalopt -ipsccp -deadargelim -instcombine -simplifycfg -basiccg -prune-eh -inline-cost -functionattrs -sroa -early-cse -lazy-value-info -jump-threading -correlated-propagation -tailcallelim -reassociate -loops -loop-simplify -lcssa -loop-rotate -licm -loop-unswitch -scalar-evolution -indvars -loop-idiom -loop-deletion -loop-unroll -memdep -memcpyopt -sccp -dse -adce -strip-dead-prototypes -always-inline
`O2`	-slp-vectorizer -globaldce -constmerge -barrier -loop-vectorize -gvn -inline (-always-inline)
`O3`	-argpromotion

program[1], and *miniFE* from the DOE proxy applications. *LULESH* [12] is a shock hydrodynamics proxy application developed by the ASCR ExMatEx Exascale Co-Design Center[2] to model numerical algorithms and data motion of scientific applications that solves a Sedov blast problem with analytical answers. We run *LULESH* with total aggregate number of elements equals to 91125 and 1728000 for single node and multiple node experiments, respectively. *LAMMPS* [23] is a molecular dynamics code that models an ensemble of particles in a liquid, solid, or gaseous state. The application computes Newton's equations of motion for system of interacting particles and can model atomic, polymeric, biological, metallic, granular, and coarse-grained systems using a variety of force fields and boundary conditions. We solve the Cu metallic solid with embedded atom method (EAM) potential which involves the dynamics of 32,000 atoms for 20,000 and 75,000 time steps for single node and multiple node experiments, respectively. *MCB*[3] models the solution of a heuristic transport equation using a Monte Carlo technique. The application employs typical features of Monte Carlo algorithms such as particle creation, particle tracking, tallying particle information, and particle destruction. The heuristic transport equation models the behavior of particles that are originated, and then travel with a constant velocity, scatter, and are absorbed. We run our *MCB* experiments with up to 12 million particles. *miniFE*[4] is a DOE proxy application which implements several kernels resembling implicit finite-element applications. The application assembles a sparse linear-system from the steady-state conduction equation on a brick-shaped problem domain of linear 8-node hex elements. Next, *miniFE* solves the linear-system using a simple un-preconditioned conjugate-gradient algorithm. We solve small problem size of 264 x 256 x 256 for single node and medium problem size of 784 x 768 x 768 for multi-node experiments.

IV. Experimental Results

In this section, we analyze effects of the compiler optimizations presented in Section II-C on the vulnerability of tested applications running on single and multiple nodes. We also analyze the performance vs. vulnerability trade-offs and the causal relations between compiler optimizations

[1]https://asc.llnl.gov/CORAL/
[2]http://science.energy.gov/ascr/research/scidac/co-design
[3]https://codesign.llnl.gov/mcb.php
[4]https://mantevo.org/

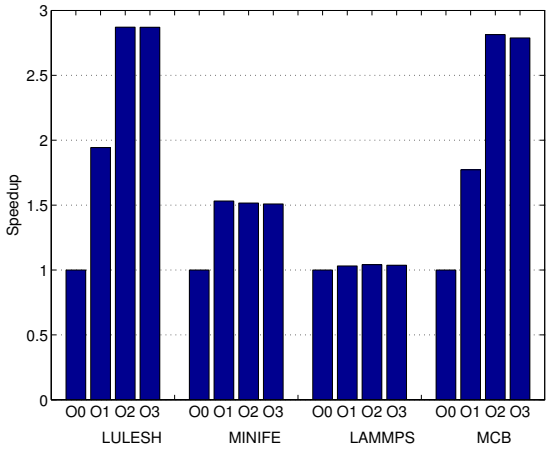


Fig. 1: Performance impact of compiler optimizations.

and application vulnerability. The experiments on single and multiple nodes are performed using the reference input set of each application as described earlier. We use all available cores for all applications except *LULESH2*, which requires a perfect cube number of MPI tasks, e.g., 27 on single node and 512 on multiple nodes. To ensure reasonable statistical significance, we conduct 1,000 runs for each application/set of compiler optimizations. We randomly select the dynamic cycle at which we inject a fault and the target MPI task. The random cycle and MPI task are extracted from two separate random sequences obtained from the same uniform random number generator using distinct seeds. We validated the random sequences with the χ^2 test and verified the uniform coverage of cycles and MPI tasks. Overall, a total of 32,000 injections were performed considering both single and multiple node experiments (1000 injections for each application, each compiled with `O0`, `O1`, `O2`, and `O3`).

A. Performance Analysis

As most compiler optimizations primarily target increased performance, we first analyze the effects of applying a different set of compiler optimizations on the applications' execution time. We apply the set of optimizations reported in Table I: increasing levels of optimization generally augment the previous set of optimizations with additional ones. For example, optimization level `O2` augments the set of optimizations used at level `O1` with `-slp-vectorizer -globaldce -constmerge`

Tab. II: Characteristics of applications and impact of compiler optimizations on performance.

	LULESH				MCB			
	O0	O1	O2	O3	O0	O1	O2	O3
Instruction decrease wrt O0	1.00	0.53	0.32	0.32	1.00	0.52	0.32	0.32
LLC-misses increase wrt O0	1.00	1.05	1.14	1.15	1.00	1.07	1.07	1.07
IPC	0.78	0.81	0.72	0.72	0.57	0.53	0.52	0.51
LLC misses/inst	0.00043	0.00084	0.00155	0.00155	0.00005	0.00010	0.00017	0.00017
Loads/inst	0.7988	0.5486	0.5414	0.5429	0.7298	0.6304	0.6260	0.6198
Stores/inst	0.0101	0.0169	0.0215	0.0215	0.0065	0.0075	0.0087	0.0083
	MiniFE				LAMMPS			
	O0	O1	O2	O3	O0	O1	O2	O3
Instruction decrease wrt O0	1.00	0.53	0.40	0.39	1.00	0.83	0.82	0.82
LLC-misses increase wrt O0	1.00	1.22	1.34	1.34	1.00	1.00	1.00	1.01
IPC	0.63	0.53	0.46	0.46	0.34	0.32	0.32	0.32
LLC misses/inst	0.00054	0.00125	0.00182	0.00183	0.00109	0.00130	0.00132	0.00134
Loads/inst	0.7722	0.5274	0.4936	0.4962	0.7779	0.6693	0.6701	0.6733
Stores/inst	0.0069	0.0128	0.0171	0.0171	0.0173	0.0207	0.0211	0.0211

`-barrier -loop-vectorize -gvn -inline`.
Figure 1 shows the speedup of applying higher compiler optimization levels with respect to optimization level `O0`. The results in the chart are average of 10 fault-free runs. As expected, increasing the compiler optimization level dramatically increases performance with speedup up to 2.7-2.8x for *MCB* and *LULESH2*, respectively. The performance improvements are much lower, but still considerable, for *miniFE* (up to 1.52x) and limited for *LAMMPS*. An interesting observation is that optimization level `O1` already provides about half of the total performance improvement. As reported in Table I, the `clang` compiler applies many important optimizations, including loop unrolling, rotating and deletion, function inlining, and memory copy optimization, at optimization level `O1`. Moving to optimization level `O2` provides considerable performance improvements for *LULESH2* and *MCB*, which benefit from vectorization, while optimization level `O3` does not vary the applications' performance.

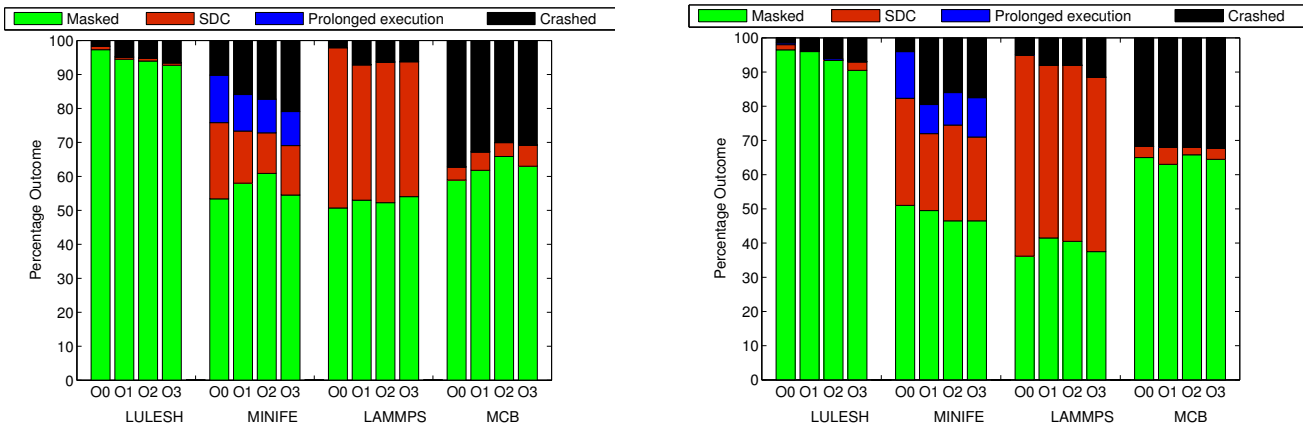
To further investigate the reasons beyond such large performance improvements, we analyze the applications' dynamic characteristics using the `perf` tools. As noted in Table II, the instructions per cycle (IPC) of most of the applications decreases when applying higher levels of compiler optimizations. However, as reported in Figure 1, higher levels of optimizations provide significant performance improvements. The reason for these performance improvements is that higher levels of compiler optimizations reduce the total number of instructions (Table II), but do not generally reduce the number of last-level cache misses, hence the Last-Level Cache misses/Instruction (LLC misses/Instr) increases and the IPC decreases. This is an interesting observation, as we expected a higher IPC when applying increasing levels of optimizations because of loop unrolling, removing unnecessary data movement, vectorization, and code elimination.

B. Vulnerability Analysis

In this set of experiments, we analyze the vulnerability of each application when applying increasing levels of compiler optimizations. We inject a single fault into one randomly selected MPI process and classify the final application outcomes according to the criteria presented in Section II-A. To understand whether an injected fault is masked or produces a SDC, we compare the applications' outputs against a fault-free outcome: the results are considered correct if they fall within 5% of the results provided by the fault-free run or if the application itself reports results to be acceptable (and vice versa). For example, *LAMMPS* reports thermodynamic state every few timestamps, and this can compared against a fault-free run to be within acceptable tolerance levels. On the other hand, *MCB* reports the Monte Carlo maximum error at the end of the run, which should be approximately equal to $1/\sqrt{Number - of - Particles}$ for the solution to be acceptable.

Figure 2a shows the percentage of cases in each category for single-node experiments with 32 cores. As we can see, each application shows a different vulnerability profile: for *LULESH2* most of the injected faults are masked (97.3%) and do not generally produce corrupted outputs ($< 3\%$) nor require additional iterations to converge ($< 1\%$). However, the injected faults may result in the application to crash. For the other applications we observe a much lower percentage of masked faults (between 50 and 65%) and a predominance of other outcomes. In particular, *miniFE* and *MCB* present a considerable number of crashes (up to 20% and 40%, respectively), while *LAMMPS* experiences a large number of cases in which the injected faults corrupt the final output. Note that *LAMMPS* and *MCB* run for a fixed number of iterations. Additionally, application developers in *LAMMPS* have provisioned built-in mechanisms to detect extreme erroneous behavior, upon which user-driven invocation of `MPI_Abort()` is done resulting in an application crash. Such cases are categorized as *SDC* in our experiments, as we assume that these mechanisms would not be present widely in most applications. Similar mechanisms are enforced in *LULESH2* and are treated in the same manner.

When increasing the level of optimization from `O0` to `O3` we notice that the number of crashes generally increases, with the exception of *MCB*. For instance, *LULESH2* demonstrates percentage increases of 195%, 221%, and 303% for `O1`, `O2`, and `O3`, respectively as compared to `O0` in the single-node experiments. Interestingly, this increase in crashes does not necessarily reduce the percentage of masked faults, at least for *miniFE* and *LAMMPS*. Rather, we observe fewer cases



(a) 1 Node (32 cores)

(b) 16 Nodes (512 cores)

Fig. 2: Statistical breakdown of application vulnerability to injected faults.

that require additional iterations to converge for *miniFE* and fewer *SDC* cases for *LAMMPS*. Overall, however, our results suggest that compiler optimizations have a stronger impact on *LULESH2* and *miniFE* than on the other two applications, as the former show a number of crashes that considerably increases with the optimization level. For *LAMMPS*, we notice that the number of crashes increases between `O0` and `O1` (relative increase of 239%), but then remains constant.

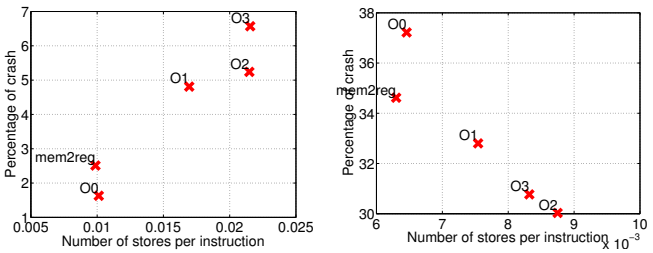
The trends observed on single-node experiments can be also identified on the multi-node experiments (Figure 2b): in both cases, the number of crashes for *LULESH2* and *miniFE* increases with the optimization levels (percentage increases up to 367% and 383% as compared to `O0`, respectively), but in the multi-node experiments the number of crashes for *LAMMPS* steadily increases with each optimization level. Comparing Figures 2a and 2b we observe a general increase in the applications sensitivity to faults, which is due to a combination of factors. First, a fault injected into a specific MPI process propagates faster in the application when the number of MPI processes increases. This is due to the higher number of MPI messages exchanged, which increases the probability that a specific MPI process contaminates others. Second, assuming a constant per-process crash probability, using more MPI processes increases the probability that the entire application crashes as the result of any of the MPI processes crashing. Notably, for *LAMMPS* with optimization level `O3`, the *Masked* cases decrease from 54.1% to 37.5% when increasing the number of MPI tasks from 32 to 512.

C. Analysis of the Causal Relation between Code Optimization and Vulnerability

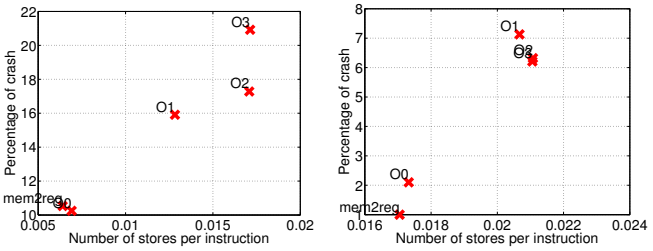
In Section IV-A, we observed that increasing levels of compiler optimizations have the potential of dramatically increasing application performance, with speedup up to 2.8x. In Section IV-B, instead, we observed that compiler optimizations also have an impact on application vulnerability and that this impact is usually negative, i.e., the vulnerability of applications increases with increasing levels of code optimization. In this section, we analyze the reasons why code optimizations impact fault masking.

As reported in Table II, the performance observed in Figure 1 is mostly achieved by applying optimizations which result in a decrease in the number of instructions, despite the fact that the IPC of the tested applications generally reduces when increasing the optimization level. In fact, LLVM applies aggressive dead code elimination (`-adce`) and combining redundant instructions (`-instcombine`) across all optimization levels. By observing the results in Table II and the results in Figure 2, we note that IPC and application vulnerability seems inversely proportional, i.e., when the IPC decreases the application vulnerability increases. We further investigate the causal relation between compiler optimizations, IPC and application vulnerability in two directions: first, we analyze the relation between stores and vulnerability and then the relation between loads and vulnerability.

The number of expected faults during an application run is greatly affected by the application execution time. However, the fact that an application runs for a shorter amount of time than another application does not imply that the former is more reliable than the latter [9]. To avoid effects of the bias induced by the different amount of instructions and execution time of each application, we analyze loads and stores with respect to the total number of instructions. Figure 3 shows the percentage of application crashes as function of the number of stores/instruction, which increases with the level of code optimization. As we can see, there is a positive correlation, albeit not perfect, between the percentage of crashes and the number of stores per instruction for all applications except *MCB*. This is due to the probability of a fault to propagate in the application memory state and, eventually, crash the application. In fact, although we inject a single bit-flip during the execution of the application, fault propagates and corrupts the application state through store instructions resulting in multiple errors/failure [1]. However, there is a probability

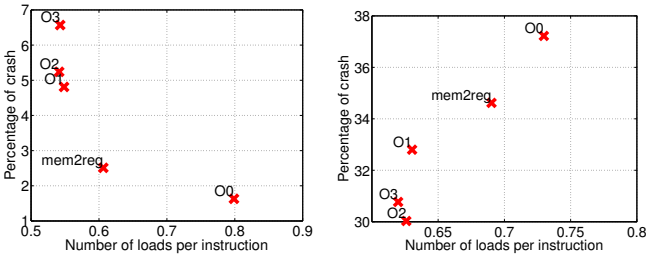


(a) LULESH (b) MCB

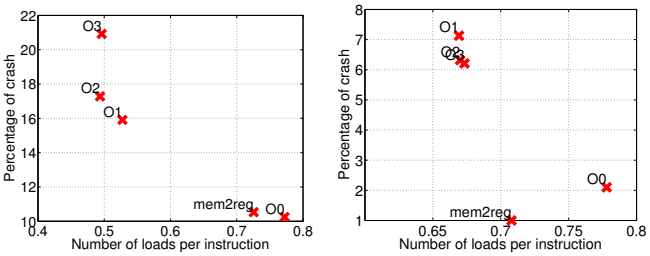


(c) MiniFE (d) LAMMPS

Fig. 3: Correlation between application vulnerability and number of stores/instruction.



(a) LULESH (b) MCB



(c) MiniFE (d) LAMMPS

Fig. 4: Correlation between application vulnerability and number of loads/instruction.

that a fault injected into a register is masked before the next store. If there is a large number of instructions between the instruction at which the fault is injected and the next store to memory, i.e., a low stores/instruction ratio, the probability of fault masking is high, hence the probability of crashes is low (and vice versa). Figure 3 confirms this observation for all applications, with the exception of *MCB*. As reported earlier in Figure 2, *MCB* does not follow any particular trend when increasing the level of compiler optimization.

We also explore the relation between the number of loads per instruction, which decreases with higher levels of compiler optimization, and the application vulnerability. In this case, a decrease in the number of loads/instruction decreases the probability that a memory load operation overwrites a corrupted register before it propagates to memory, hence masking the injected fault. Figure 4 shows that, indeed, as the number of loads/instruction decreases, the percentage of crashes decreases.

The correlations between loads/instruction and stores/instruction and the number of crashes resulting from the fault injection experiments indicate the intrinsic vulnerability of each application, i.e., the ability of an application to mask an eventual fault that occurs at register level. All applications except *MCB* follow the same trend, i.e., a high stores/instruction ratio or a low loads/instruction ratio indicate high numbers of crashes. Notably, *MCB* has a significantly lower number of stores/instruction as compared to other applications even with optimization level `O0`, which may be one of the reasons of its distinct behavior. In particular, it appears that each store instruction has a higher probability of corrupting loop control variables, as shown in the next section. We attribute this characteristics to

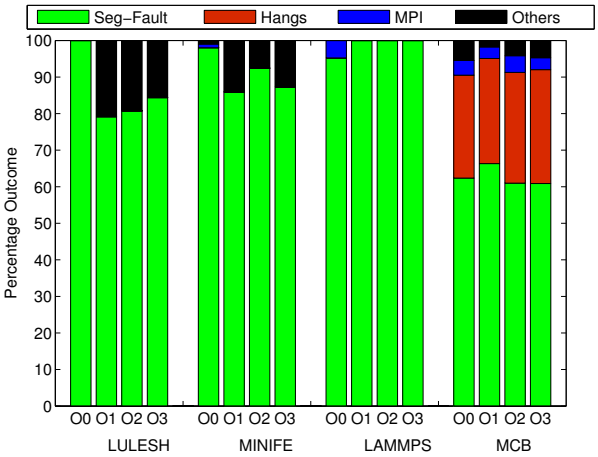


Fig. 5: Crash breakdown.

the random sampling of the Monte Carlo algorithm, which produces a less regular computation and memory access pattern [21].

D. Crash Analysis

Finally, we analyze the *Crash* cases in more detail to gain insights into the different causes that may produce an abrupt termination of an application. The results of this analysis are presented in Figure 5. At compiler optimization level `O0`, most of the crashes are due to segmentation faults caused by attempts to access a restricted or not allocated portion of the application address space, probably as the result of corrupting a register that stores the value of a pointer. With increasing compiler optimizations, some other reasons for *Crash* also arise, such as "MPI faults" caused by erroneous inputs to MPI

Tab. III: Application Vulnerability Characteristics.

Bench	Class	Prol. Exec.	Vulner.	Converg.
LULESH	Sedov Blast Problem	Yes	Low	High
MiniFE	Finite-Element method	Yes	Medium	Low
LAMMPS	Molecular Dynamics	No	Medium	N/A
MCB	Transport Equation	No	High	Low

routines, e.g., an erroneous destination rank in a `MPI_Send` function call (note, that no faults are injected in the MPI library). The "hangs" cases observed for *MCB* are due to the application incurring in a time out (set to 4x the execution time of the fault-free version). Figure 5 shows that a considerable number of crashes are due "other" errors: these are mainly raised by the memory controller when attempting to access erroneous physical addresses.

E. Discussion and Future Directions

Overall, the results indicates that highly-optimized code is more vulnerable than less-optimized code, but also there is a good chance that a fault may be masked before propagating to memory, either by a register refresh or by other arithmetic operations, provided that there is "enough time" before the store operation to memory. In the same way as the performance impact of compiler optimization depends on the application, our results show that the impact on application vulnerability also depends on the particular application and algorithm. Table III is an effort to summarize the characteristics of each application and a qualitative assessment of the general vulnerability of the parallel applications tested in this work, as resulting from our experiments. Most of the HPC applications are iterative and arithmetically-intensive in nature, thus there is a chance that the fault is either masked or propagated to later iterations. Moreover, the convergence properties of each applications play an important role in the detection and masking of SDC.

Interesting trade-offs between vulnerability and performance are found for multiple applications from the results in figures 1 and 2. For example, *LULESH2* achieves 1.9x and 2.8x speedup with optimization `O1` and `O2`, respectively. Whereas, there is only negligible performance benefit of moving on to `O3`. On the vulnerability front, the (absolute) percentage of crashes are 4.8%, 5.2% and 6.6% for `O1`, `O2`, and `O3` respectively, compared to 1.6% at `O0`. In this case, there is a notable increase in vulnerability moving on from `O2` to `O3`, with virtually no performance benefit. On the other end, there is significant performance benefit of moving on from `O1` to `O2`, with only slight increase in vulnerability. Similarly, for *miniFE*, using `O2` and `O3` does not bring extra performance benefit with respect to `O1`, but the vulnerability keeps increasing with each optimization level. More importantly, our results indicate that blindly increasing the level of code optimization without considering the effects on the application vulnerability might not be a wise choice, especially in the context of future exascale systems where these numbers are projected to be higher. This observation suggests that the compiler cost function should account for vulnerability, as well as code size and performance, and indicate optimizations that only apply 'safe' code transformations. As future work, an intelligent runtime system could leverage automatic techniques, such as genetic algorithms, to traverse the performance-vulnerability optimization search space [11]. A recent work [19] has demonstrated some positive results by use of genetic algorithm-based technique to find a sequence of optimizations at compile-time which lowers the vulnerability of the application as compared to unoptimized version. This study uses applications from PARSEC and Parboil benchmark suites, and it would be interesting to apply the same technique for HPC applications at scale as part of future work. The results in this paper show that such a study would be worthwhile.

Moreover, our results do not indicate any significant variation in the number of SDC cases with increase of optimization levels. In fact, for most applications, there is a slight decrease in SDC cases. For instance, *LAMMPS* demonstrates about 15% (relative) decrease with both `O1` and `O3`, as compared to `O0`. On the other hand, slight increases in masked cases of about 6.4%, 2.1%, and 6.2% (relative) are noted for *MCB*, *miniFE*, and *LAMMPS* at `O3`, respectively, as compared to `O0`. Whilst, for *LULESH*, a relative decrease of 4.9% in masked cases is noted at `O3` as compared to `O0`. Overall, no discernible trend is visible for both SDC or Masked scenarios (see Figure 2 for more details) in comparison to the effect on crashes. Thus, there is nothing significant to report in this forefront. Albeit, this could be an artifact of the adopted fault model. Extending our fault model to include DRAM faults and multi-bit faults [32] in future work, can provide decisive conclusions about the impact of compiler optimizations on SDCs, if any.

V. Related Work

Application vulnerability studies provide important insights into application behavior and the proper mechanisms that should be employed to reach the desired level of application performance and/or resiliency. Critical application points identified using such studies, can then be hardened against soft errors by techniques such as instruction duplication to ensure correct application state is maintained during execution or otherwise the error is detectable [26]. Software-only techniques, such as *software implemented fault tolerance (SWIFT)*, reduce the overhead of instruction duplication through control-flow checking and exploiting unused instruction-level parallelism [24]. For instance, reliability-driven software compilation technique for SWIFT is proposed in [28]. A detailed comparison of hardware/software techniques for soft-error protection of embedded processors is presented in [15]. In this work, we investigate whether compiler-based software-only techniques can effect the resiliency of distributed HPC applications in the presence of soft errors, especially due to their significance in future large scale system design.

Compiler Optimization & Application Vulnerability: The impact of compiler optimizations on the vulnerability of

applications has been studied in other application domains [8], [9], [31], [25], [33], [35], [19], but not extensively in the HPC domain. In [9], the effects of the occupancy of various micro-architecture structures such as the reorder buffer, the instruction fetch queue and load store queue as a result of different compiler optimizations are studied. The authors also propose the Expected Failure metric and show that optimized code is more reliable according to their failure metric which is proportional to execution time. However, if the unoptimized code was to run for equivalent time, then it would be more reliable than optimized code. Similarly, speculatively-scheduled loops are shown to be more vulnerable than unrolled loops using the PVF metric [31]. LLFI has been used to quantify the effects of compiler optimizations for soft computing applications [33]. The authors focus primarily on egregious data corruptions, where the resulting loss in signal-to-noise ratio for image processing applications is greater then 30 dBs (12% of the overall reported SDCs). Despite large amount of work in other domains, the impact of compiler optimizations on vulnerability of HPC applications has not been explored.

Vulnerability metrics: Different set of metrics have been proposed to assess the vulnerability of an application. The *architectural vulnerability factor* (AVF) is based on the susceptibility of micro-architecture components inside of a processor [18]. In this case, the probability of an application failure is determined when a fault affects a particular micro-architecture component. Whereas, the PVF metric studies the effects of application vulnerability independently of the hardware components and can be measured with fault injection or architecturally correct execution [31]. The *data vulnerability factor* (DVF) [40] is based on memory access pattern and failure rate. Using DVF, it is shown that preconditioned conjugate gradient is more vulnerable than conjugate gradient for smaller workloads.

Fault Injection: The goal of fault injection studies is to determine the applications' vulnerability to faults at various abstraction layers, ranging from circuit [27], to architectural [5], [2], [38], to compiler and application level [30], [39], so that appropriate fault tolerance mechanisms can be employed at those layers. An extensive treatment on fault-propagation from the flip-flops in the circuit-level to the application-level [7] is beyond the scope of this work. Herein, we extended LLFI to study the vulnerability of MPI applications by injecting single fault into independent MPI processes.

VI. Conclusions

The impacts of compiler optimizations on application performance have been widely studied in the past. However, as we approach the exascale era, it can be worthwhile to understand the new trade-offs between application performance and vulnerability and how code optimizations impact the vulnerability of HPC applications. Outcomes of injecting single-bit faults in DOE applications on single- and multi-node environments show that, as for performance improvements, the impact of compiler optimizations on application vulnerability strongly depends on the application structure and algorithm. Some applications are inherently more robust than others and can tolerate partially-corrupted memory states, possibly at the cost of increased execution time. We also analyzed the causal relations between compiler optimizations and the applications' vulnerability and showed that highly-optimized code is usually more vulnerable than unoptimized code. On the other hand, we show that certain cases of optimization may provide limited performance improvement that may not pay off due to the increase in the application vulnerability.

Acknowledgment

This work was supported in part by the DOE Office of Science, Advanced Scientific Computing Research, under award number 59542 "Performance Health Monitor"; program manager Lucille T. Nowell.

References

[1] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in HPC applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015, pp. 72:1–72:12.

[2] R. Balasubramanian, Z. York, M. Dorran, A. Biswas, T. Girgin, and K. Sankaralingam, "Understanding the impact of gate-level physical reliability effects on whole program execution." in *the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, February 2014.

[3] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of dram raw error rate on a supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, 2016, pp. 55:1–55:11.

[4] C. Bender, P. Sanda, P. Kudva, R. Mata, V. Pokala, R. Haraden, and M. Schallhorn, "Soft-error resilience of the IBM POWER6 processor input/output subsystem," *IBM Journal of Research and Development*, vol. 52, no. 3, 2008.

[5] S. Bohm and C. Engelmann, "xSim: The extreme-scale simulator," in *The Int. Conference on High Performance Computing and Simulation (HPCS)*, July 2011.

[6] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller, "Understanding soft error resiliency of BlueGene/Q compute chip through hardware proton irradiation and software fault injection," in *Int. Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC, 2014.

[7] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *50th ACM/EDAC/IEEE Design Automation Conference*, May 2013.

[8] J. J. Cook and C. B. Zilles, "A characterization of instruction-level error derating and its implications for error detection," in *the International Conference on Dependable Systems and Networks (DSN)*, 2008.

[9] M. Demertzi, M. Annavaram, and M. Hall, "Analyzing the effects of compiler optimizations on application reliability," in *IEEE Internation Symposium on Workload Characterization*, Nov 2011.

[10] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253–266, Feb 2010.

[11] N. Imran, R. A. Ashraf, and R. F. DeMara, "Power and quality-aware image processing soft-resilience using online multi-objective GAs," *Int. J. Comp. Vision Rob.*, vol. 5, no. 1, Jan 15.

[12] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium*, 2013.

[13] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, and et al., "Exascale computing study: Technology challenges in achieving exascale systems," DARPA IPTO, Tech. Rep. DARPA-2008-13, September 2008.

[14] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC, Salt Lake City, Utah, 2012.

[15] A. Martinez-Alvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F. Pinto, H. Guzman-Miranda, and M. Aguirre, "Compiler-directed soft error mitigation for embedded systems," *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 2, pp. 159–172, March 2012.

[16] S. Michalak, A. DuBois, C. Storlie, H. Quinn, W. Rust, D. DuBois, D. Modl, A. Manuzzato, and S. Blanchard, "Assessment of the impact of cosmic-ray-induced neutrons on hardware in the roadrunner supercomputer," *Device and Materials Reliability, IEEE Trans. on*, vol. 12, no. 2, June 2012.

[17] S. Mitra, P. Bose, E. Cheng, C.-Y. Cher, H. Cho, R. Joshi, Y. M. Kim, C. R. Lefurgy, Y. Li, K. P. Rodbell *et al.*, "The resilience wall: Cross-layer solution strategies," in *VLSI Technology, Systems and Application (VLSI-TSA), Proceedings of Technical Program-2014 International Symposium on*. IEEE, 2014, pp. 1–11.

[18] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th IEEE/ACM Int. Symposium on Microarchitecture*, ser. MICRO 36, 2003.

[19] N. Narayanamurthy, K. Pattabiraman, and M. Ripeanu, "Finding resilience-friendly compiler optimizations using meta-heuristic search techniques," in *2016 12th European Dependable Computing Conference (EDCC)*, Sept 2016, pp. 1–12.

[20] N. Narayanamurthy, "Finding resilience-friendly compiler optimizations using meta-heuristic search techniques," Ph.D. dissertation, University of British Columbia, 2015. [Online]. Available: https://open.library.ubc.ca/cIRcle/collections/24/items/1.0166741

[21] R. Neely, "Proxy applications: Vehicles for co-design and collaboration," Dec. 2013, presented at Predictive Science Academic Alliance Program (PSAAP) II Meeting.

[22] R. B. Parizi, R. R. Ferreira, L. Carro, and Á. F. Moreira, *Compiler Optimizations Do Impact the Reliability of Control-Flow Radiation Hardened Embedded Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 49–60.

[23] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. of Computational Physics*, vol. 117, no. 1, pp. 1 – 19, 1995.

[24] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, March 2005.

[25] B. Sangchoolie, F. Ayatolahi, R. Johansson, and J. Karlsson, "A study of the impact of bit-flip errors on programs compiled with different optimization levels," in *European Dependable Computing Conference*, May 2014.

[26] S. Sastry Hari, S. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Application resiliency analyzer for transient faults," *IEEE Micro*, vol. 33, no. 3, pp. 58–66, May 2013.

[27] N. Seifert and N. Tam, "Timing vulnerability factors of sequentials," *Device and Materials Reliability, IEEE Transactions on*, vol. 4, no. 3, pp. 516–522, Sept 2004.

[28] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel, "Exploiting program-level masking and error propagation for constrained reliability optimization," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13, 2013, pp. 17:1–17:9.

[29] V. C. Sharma, G. Gopalakrishnan, and S. Krishnamoorthy, "Towards resiliency evaluation of vector programs," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1319–1328.

[30] V. C. Sharma, A. Haran, Z. Rakamarić, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2013.

[31] V. Sridharan and D. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, Feb 2009.

[32] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," in *Proceedings of 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2015.

[33] A. Thomas, J. Clapauch, and K. Pattabiraman, "Effect of compiler optimizations on the error resilience of soft computing applications," in *Workshop on Algorithmic and Application Error Resilience (AER)*, June 2013.

[34] A. Tiwari and J. Torrellas, "Facelift: Hiding and slowing down aging in multicores," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41, 2008, pp. 129–140.

[35] L. Wang, R. Bertran, A. Buyuktosunoglu, P. Bose, and K. Skadron, "Characterization of transient error tolerance for a class of mobile embedded applications," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, Oct 2014, pp. 74–75.

[36] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," in *Proceedings of 34th Annual International Symposium on Computer Architecture*, ser. ISCA, 2007, pp. 460–469.

[37] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *International Conference on Dependable Systems and Networks*, 2004.

[38] Z. Wang, C. Chen, and A. Chattopadhyay, "Fast reliability exploration for embedded processors via high-level fault injection," in *14th International Symposium on Quality Electronic Design*, March 2013, pp. 265–272.

[39] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *IEEE International Conference on Dependable Systems and Networks*, June 2014.

[40] L. Yu, D. Li, S. Mittal, and J. S. Vetter, "Quantitatively modeling application resilience with the data vulnerability factor," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014, pp. 695–706.