Declarative Resilience: A Holistic Soft-Error Resilient Multicore Architecture that Trades off Program Accuracy for Efficiency

HAMZA OMAR, QINGCHUAN SHI, MASAB AHMAD, HALIT DOGAN, and OMER KHAN, University of Connecticut, USA

To protect multicores from soft-error perturbations, research has explored various resiliency schemes that provide high soft-error coverage. However, these schemes incur high performance and energy overheads. We observe that not all soft-error perturbations affect program correctness, and some soft-errors only affect program accuracy, i.e., the program completes with certain acceptable deviations from error free outcome. Thus, it is practical to improve processor efficiency by trading off resiliency overheads with program accuracy. This article proposes the idea of declarative resilience that selectively applies strong resiliency schemes for code regions that are crucial for program correctness (crucial code) and lightweight resiliency for code regions that are susceptible to program accuracy deviations as a result of soft-errors (non-crucial code). At the application level, crucial and non-crucial code is identified based on its impact on the program outcome. A cross-layer architecture enables efficient resilience along with holistic soft-error coverage. Only program accuracy is compromised in the worst-case scenario of a soft-error strike during non-crucial code execution. For a set of machine-learning and graph analytic benchmarks, declarative resilience reduces performance overhead over a state-of-the-art system that applies strong resiliency for all program code regions from ~ 1.43× to ~ 1.2×.

CCS Concepts: • **Computer systems organization** → Multicore architectures; Resilience; Redundancy;

Additional Key Words and Phrases: Program accuracy, soft-errors, graph analytics, machine learning

ACM Reference format:
Hamza Omar, Qingchuan Shi, Masab Ahmad, Halit Dogan, and Omer Khan. 2018. Declarative Resilience: A Holistic Soft-Error Resilient Multicore Architecture that Trades off Program Accuracy for Efficiency. *ACM Trans. Embed. Comput. Syst.* 17, 4, Article 76 (July 2018), 27 pages.
https://doi.org/10.1145/3210559

1 INTRODUCTION

The ever-increasing miniaturization of semiconductors has led to important advancements in mobile, cloud, and network computing. However, it has caused electronic devices to become less reliable and microprocessors more susceptible to transient faults induced by radiations. These intermittent faults do not provoke permanent damage; however, they may result in incorrect execution

This research was partially supported by the National Science Foundation under Grant No. CCF-1550470.
Authors' addresses: H. Omar, Q. Shi, M. Ahmad, H. Dogan, and O. Khan, 371 Fairfield Way, Unit 4157, Storrs, CT, 06269, USA; emails: {hamza.omar, qingchuan.shi, masab.ahmad, halit.dogan, omer.khan}@uconn.edu.

https://doi.org/10.1145/3210559

of programs by altering signal transfers or stored values. These transitory faults are also called soft errors. As technology scales, researchers and industry pundits are projecting that soft-errors will become increasingly important [14]. Today's processors implement multicores, featuring diverse set of compute cores and on-board memory sub-systems connected via networks-on-chip and communication protocols. Such multicores are widely deployed in numerous environments for their computational capabilities, from traditional applications such as data centers to emerging areas including unmanned aerial vehicles (UAVs) [49] and self-driving cars [23]. These cyber-physical systems require high resilience for safety-criticality [66, 68], yet high performance for their timing constraints. Applications running on such systems include graph analytics (e.g., path planning, motion detection), computer vision, and artificial intelligence (e.g., machine learning) [28, 53, 54]. The challenge is to prevent such systems from failure due to soft-errors, while still meet real-time processing constraints.

While extensive research has been done on protecting single core processors from soft-errors, multicore systems introduce new challenges, especially when running parallel applications under complex shared memory protocols. Multicores integrate compute pipelines, cache hierarchy, and interconnection networks on a single die, which introduces additional challenges that lead to complex logic interactions and makes high soft-error coverage guarantee a hard problem. Moreover, shared memory complicates error detection and recovery mechanisms, since data races among cores lead to false alarms and make it harder to replay the program in a deterministic manner. The error detection-to-recovery latency suffers when many cores rollback and synchronize their redundant execution. Although these rollbacks may happen rarely, with safety-critical systems' tight real-time constrains, such recovery schemes may not be acceptable. This article's objective is to develop an efficient resilient multicore architecture with high soft-error coverage. The program execution must produce programmer acceptable result under soft-error perturbations and meet real-time constraints.

First, the existing soft-error resilience schemes are reviewed in the context of the above stated objective. A Hardware Redundant Execution (HaRE) [57] scheme was developed earlier, which relies on temporal redundancy for all program instructions. It relies on a local per-core re-execution mechanism to to detected soft-errors. Moreover, it implements a resilient coherence protocol [2] for on-chip communication. This avoids expensive global checkpoint and roll-backs, while enabling holistic protection for the multicore system. It also ensures the feasibility of switching the re-execution on/off at each core, since one core's re-execution does not affect instruction execution of another core. Holistic protection schemes for multicore (and GPU) systems have been developed using mechanisms, such as thread-level redundancy (TLR) [43, 69]. These schemes deliver high coverage by performing cross checks in a duplicated thread. However, redundant computation sacrifices available parallelism, which leads to relatively large performance overheads. In general, multi-threading schemes operate at coarse granularity and require complex checkpoint and global roll-backs. Thus, they require long detection-to-recovery latency, which is not ideal for systems with real-time constraints. To improve performance, the idea of selectively applying resilience protection in a program has been explored. Research has focused on applying certain n-Modular Redundancy (nMR) or symptom-based schemes selectively to a program [30, 45]. These schemes apply protection based on code's vulnerability, which makes the program less likely to be affected. However, they do not bound soft-error's impact. In similar context, the idea of lowering program accuracy for performance has been explored in approximate computing [12, 13, 37, 58, 60]. These works do not focus on the soft-error resilience perspective. Not all code in an approximated program can tolerate errors, and still requires certain level of resilience protection. Recently, Khudia et al. [22] explored the idea that selects instructions based on their program accuracy impact. Like most of the selective resilience schemes, it only protects certain instructions

Table 1. Summary of Various Resilience Schemes

	Performance Overhead	Hardware Overhead	Selectively Trading-Off	Multicore	Compiler Aid	Coverage			
						Crashing	Deadlock	Livelock	SDC
TLR [43]	High	High	None	Yes	Not Applicable	**High**	**High**	**High**	**High**
HaRE [57]	Mid	Mid	None	Yes	Not Applicable	**High**	**High**	**High**	**High**
dTune [45]	Mid	Mid	Vulnerability	Yes	Not Applicable	Mid	Mid	Mid	Mid
RASTER [30]	Mid	Mid	Vulnerability	No	Not Applicable	Mid	Mid	Mid	Mid
Khudia et.al. [22]	Low	None	Accuracy	No	Yes	Low	Low	Low	**High**
Declarative Resilience	Low	Mid	Accuracy	Yes	Possible	**High**	**High**	**High**	**High**

"High/Mid/Low" in fault type refers to the capability of protecting system from such faults.

and leaves others unprotected, and focuses on Silent Data Corruption (SDC). Moreover, it is not specifically designed for multicores. The proposed declarative resilience architecture for shared memory multicores applies different resilience schemes to different code regions based on their criticality. Hardware Redundant Execution (HaRE) scheme is used for the crucial code regions that affect program correctness. Lightweight schemes are used for the non-crucial code regions, where compromising program accuracy is acceptable in case of soft-errors. It reduces performance overhead of resiliency by eliminating unnecessary redundant executions for non-crucial code regions.

Various state-of-the-art resiliency schemes are summarized in Table 1 and contrasted to the proposed declarative resilience scheme in terms of performance, hardware overhead, selective resiliency, and coverage tradeoffs. The terms "High/Mid/Low" are used in Table 1 categorize the capability of protecting systems from soft-errors. "High" means that in most cases the error can be prevented or detected/recovered. "Mid" implies the scheme is only effective in some cases, and the fault effects are still commonly visible. Lastly, "Low" shows that it is unlikely the system can be protected under such faults. Note that for all discussed resilience schemes, exceptions are managed by the operating system. It is evident from Table 1 that the proposed declarative resilience architecture provides high coverage, as well as adjusts the program accuracy to achieve better performance with minimal hardware overheads.

1.1 Organization

The article is organized as follows. Section 2 discusses the proposed declarative resilience architecture and the design flow necessary to integrate various resiliency mechanisms. Moreover, it explains the process of classifying application code among crucial and non-crucial regions. Section 3 explains the applications under consideration, and discusses potential non-crucial regions to achieve performance and energy benefits. It also provides description on accuracy threshold selection. The methodology is reviewed in Section 4. Section 5 explains the accuracy, performance, and energy results reported for all applications under the proposed architecture. Section 6 discusses prior work in the field of resilience, and approximate computing. Last, the article is concluded in Section 7.

2 DECLARATIVE RESILIENCE

The key idea of declarative resilience architecture is to protect different code regions with different resilience schemes that trade-off program accuracy with efficiency. The novelty comes from

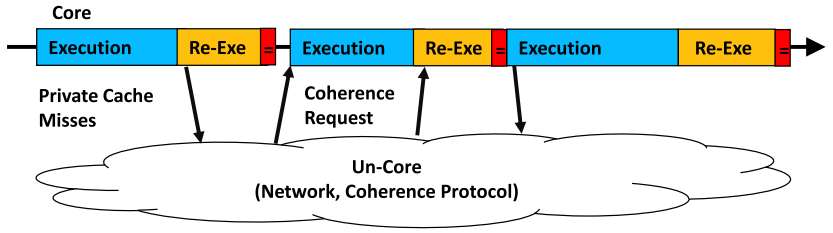

Fig. 1. Hardware resilience mechanism (HaRE) protects each compute core using redundant execution and un-core using a resilient cache coherence protocol.

two factors: the way code regions are identified and the resilience schemes to ensure no side-effects due to soft-errors. Based on the notion of trading off resilience overheads with program accuracy, crucial and non-crucial code is defined as follows. Crucial code affects program correctness, which means the program should be able to complete without crashing, deadlocking, and so on, due to soft-errors, while its outcome is explicable. Non-crucial code only affects program accuracy, which refers to how much the result is off compared to error-free scenario. In the proposed architecture, we use Hardware Redundant Execution scheme [57] (HaRE) for crucial code and lightweight Software-Hardware Resiliency schemes (SHR) for non-crucial code.

Recent research [14, 17, 22] has pointed out that SDC can prove to be the most harmful effect of soft-error, since SDCs do not produce obvious failures. This limits the user to know whether the results are correct or not. As shown in Table 1, the proposed architecture also targets high coverage for SDC in addition to program crashing, deadlock, and live-lock type of errors. Moreover, this article considers soft-errors that impact the program control flow or data flow. The OS (operating system) code is not emulated, and exceptions (including ones triggered by soft-errors) must be handled explicitly by the OS. In the remaining sections, we introduce the hardware resilience mechanism (HaRE) for protecting crucial code, then we explain how to select non-crucial code regions based on soft-error's effects at instruction level and the resilience protection needed to ensure program correctness. Next, we go over the design flow to classify code regions as crucial versus non-crucial and discuss systematic assist and certain optimizations. An application illustration using a machine-learning workload is presented at the end.

2.1 Architectural Support for Redundant Execution

For crucial code regions, a state-of-the-art hardware re-execution based resilience mechanism (HaRE) is proposed that guarantees high soft-error coverage [57]. In HaRE (as shown in Figure 1), each core re-executes its own atomic instruction sequences, and rollbacks to a safe state when soft-errors are detected. For deterministic and deadlock-free re-execution, HaRE guarantees atomicity for instruction sequences: modified data is not committed or transferred until control and data flow is checked using hardware signature registers. It has two main phases: *execution* and *redundant execution (re-execution)*. At the beginning of the execution phase, all necessary states such as register file and program counter contents are duplicated to ensure a safe state checkpoint. Re-execution is triggered when the data needs to be consumed, such as when another core sends a coherence request (e.g., invalidation of a cache line). Private cache misses also trigger re-execution to optimize performance, since an in-order core would likely be idle for tens of cycles while waiting for the reply. Upon the triggers (overall there are seven trigger events in HaRE), the instruction sequence is re-executed and checked to ensure that all the data leaving the core is correct. Soft-error perturbations can be found by comparing a set of control and data-flow signature registers

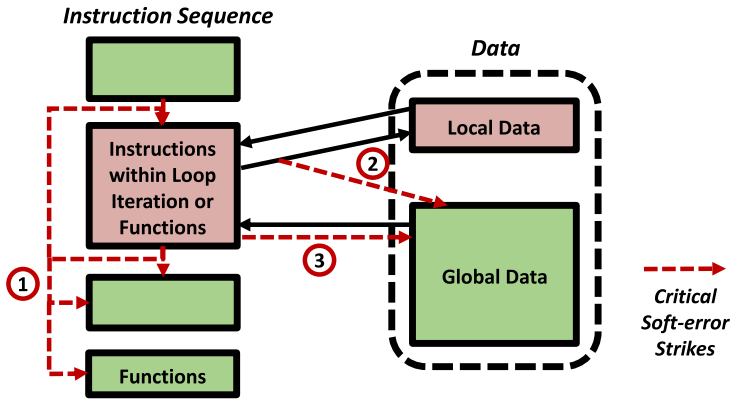

Fig. 2. Soft-errors' effects to program control and data flows.

captured during both execution and re-execution phases. In case of a mismatch, another re-execution is used to recover. A resilient cache coherence protocol [2] is used to enable protection for the on-chip communication hardware. Overall, HaRE can exploit core level locality for re-execution and can trigger it on long latency stalls to hide the overhead. Since HaRE is implemented at the hardware level, the execution and re-execution phases are transparent to the programmer. As HaRE re-executes all instructions, its performance can be improved by only applying re-execution to certain code regions. HaRE is suitable for turning re-execution on or off at instruction level, because it is applied locally to each core. However, since each transition has a latency cost, the challenge is to develop a programmer-assisted mechanism to identify "code regions" that have limited effects on program outcome under soft-error perturbations and efficiently pass it to the hardware.

2.2 Guidelines of Non-Crucial Instructions

In the proposed declarative resilience architecture, all instructions are assumed crucial until proven otherwise. To identify the non-crucial code regions, let us first analyze the soft-error effects at instruction level. The control flow includes branches, loops, jumps, and function calls, as shown in Figure 2. A soft-error can affect the control flow instruction in two ways: wrong target/return address, or wrong branch condition. Wrong target/return address can result in accessing arbitrary data and cause unpredictable effects, as shown in Figure 2(①). Moreover, soft errors can impact the program counter (PC), which can lead to some other instruction being executed. Thus, control flow instructions and the calculations of their target/return addresses are always crucial. For example, when HaRE is enabled around the "for" statement at program level, calculations of its target/return addresses are automatically protected. Moreover, a conditional control flow instruction may incorrectly calculate its branch condition under soft-errors. Loop statements, such as "for" and "while," as well as the updates to their loop counter variables, are always protected for the applications evaluated in this article. Wrong branch conditions could potentially affect program correctness. However, these conditions could be tolerated if the effects reside only in the non-crucial region and impact program accuracy. Consider an example where a soft-error strike is subjected to a branch condition in the non-crucial code. If the branch decides among two different data flow instructions, then choosing either condition will only affect program accuracy.

In program data flow, as store instructions directly modify memory and make data visible to the program, they are more important than other instructions. As shown in Figure 2(②), a store

instruction can modify data at an arbitrary memory location unexpectedly due to incorrect store address. Also, a store instruction can leave data unchanged, whereas it was intended to be updated. Thus, store address calculation including preceding instructions in case of indirect addressing are considered crucial. Store instructions can also affect program outcome through data committed to memory. To illuminate the commit process, data is divided into two categories: *local* and *global*, as shown in Figure 2. Local data is only used within a certain code region. It is also temporal, because it is not used after exiting the region, for example, variables defined or initialized within a loop iteration or a function. Data that is consumed outside the region is considered global. Hence, local data is considered non-crucial, and global data can be crucial. However, local data is accumulated to global data through computations. Values accumulated to global data are considered crucial, as shown in Figure 2(③). Thus, computations of local data may also need to be protected for program correctness.

Based on our insights about soft-error effects on program control and data flow, the ideal candidates for non-crucial instructions are *compute instructions* in-between control flow that only execute on local data and have minimum impact on program outcome. One example would be random number generation instructions in Monte Carlo method. However, in real applications instructions that meet all these conditions are rare. In addition, turning HaRE on and off introduces overheads that must be amortized by lowering the frequency of switching between crucial and non-crucial instructions. Next, a set of lightweight software and hardware resilience mechanisms are introduced that enable the proposed architecture to compose non-crucial code regions.

2.3 Non-Crucial Code Regions

As mentioned earlier, the frequency of switching HaRE on/off has detrimental effects on performance. Therefore, for declarative resilience to be beneficial, it is favorable to have a set of contiguous non-crucial instructions to hide HaRE on/off switching overheads. Such continuous regions can be obtained using *loop-unrolling*. It is a loop transformation technique that attempts to optimize the execution speed of the program by reducing instructions that control the loop, such as *end of loop* tests on each iteration. Applying *loop-unrolling* optimization can potentially result in performance gains. However, it involves program loops to be re-written as a repeated sequence of similar independent commands and statements. This allows the soft-error effects on the program outcome to be restricted for certain code regions. For this purpose, a set of lightweight software and hardware resilience (SHR) mechanisms are introduced on top of HaRE.

First, at the hardware level, SHR applies protection for store instructions. Based on the insight that store addresses are always critical to the program, SHR performs hardware-level redundant address calculations. These calculations incur additional overheads. Store operations proceed after their address calculation is verified. Otherwise, they are re-executed. This is to ensure that store instructions do not access the crucial region unexpectedly. Second, at the software level, the value committed to global data is checked to ensure program correctness and accuracy. This bound checking is critical, since non-crucial code can affect program response if the value committed to memory is not checked. For example, consider a simple program P that increments the value of x by 1 (x++) each time for 100 iterations. For P, the bound-checker would keep track of x such that it never goes beyond 100 or is never less than 0—this could happen if the soft-error strike perturbs x to some arbitrary value. If the program fails to pass the bound-checking process, then it is re-executed. Otherwise, the program proceeds to the next step. It is practical to use a programmer defined software-level bound checker to provide certain resilience protection.

It is worthwhile noting that the proposed scheme provides high SDC coverage by using bound checkers. Since the silently corrupted data value is bounded, the program output would be restricted to diverge from an acceptable result. We claim that the accuracy of the result (corrupted

or not) would always be less than the accuracy threshold, if the bound checker is satisfied. The bounds are predefined—such as 100 in the aforementioned example—to make the worst case acceptable. Thus, SDC by itself should not cause unpredictable results.

With SHR, store instructions and local data computations with predictable outcome are able to be included in non-crucial code regions. This makes it more feasible to cluster contiguous non-crucial instructions into code regions. In summary, code regions are classified as non-crucial with SHR if they do not contain control flow instructions, and meet one of the following:

- They do not access global data.
- The value committed to global data does not affect the program response.
- The value committed to global data can be bound checked.

In addition to the above rules, there are multiple scenarios that require careful analysis. Such scenarios are outlined below:

(1) ***The opcode is perturbed.*** Soft-errors may perturb the opcode of an assembly instruction causing a data flow instruction to become a control flow type. For example, a non-crucial compute, load, or store instruction could be perturbed to a branch instruction. As control flow instructions are not allowed to be part of non-crucial region, an exception would be triggered. For the same reason, the PC of next instruction should always increment sequentially, otherwise an exception should be triggered.

(2) ***The operand is perturbed (When no exception triggered).*** Since a load value is only used for computation in a non-crucial code region, a perturbed load operand (address) only results in wrong value for the subsequent instruction in the region. Thus, bit-flips in the operands (such as in the decode or execution stage) of compute and load instructions only result in perturbed value of temporal variables. These are checked using the bound checker, and dropped if found to be out of bound. Meanwhile, soft-errors in the operands of store instructions result in accessing arbitrary memory locations, thus the store address calculation is protected by SHR.

(3) ***When exceptions are triggered.*** In the article, we do not consider OS actions. All exceptions are to be handled as done in the baseline system. Such as in the case of a perturbed load access to out of bound address, the OS handles the triggered segmentation fault. Additional OS support is needed to handle the exception when control-flow instructions are present in non-crucial regions. The OS reports the error to the user. Upon receiving the error, more coarse grain resilience schemes can be applied, such as the user can re-execute the entire program.

Based on the above criteria, a significant amount of code in the targeted machine-learning and graph analytic programs can be considered non-crucial. Next, we describe the proposed design flow to accomplish classification of code as crucial versus non-crucial.

2.4 Design Flow and Systematic Assist

Figure 3 shows the design flow of the proposed declarative resilience architecture. Prior work [56] has been extended in various aspects to formulate a rigorous and efficient resilience framework. In the design of the proposed architecture, only the step for potential crucial and non-crucial identification of code regions is programmer dependent. All other steps are implemented with systematic assistance and need no programmer intervention.

2.4.1 Configuration Identification & Tuning. A structured and rigorous region classification process is introduced to identify and mark regions as crucial/non-crucial. The programmer is required to first identify the non-crucial instructions based on the classification guidelines (cf. Section 2.2). As mentioned earlier, it is beneficial to have contiguous compute instruction sequences within each non-crucial region. With SHR in mind, schemes such as *software level bound checking*

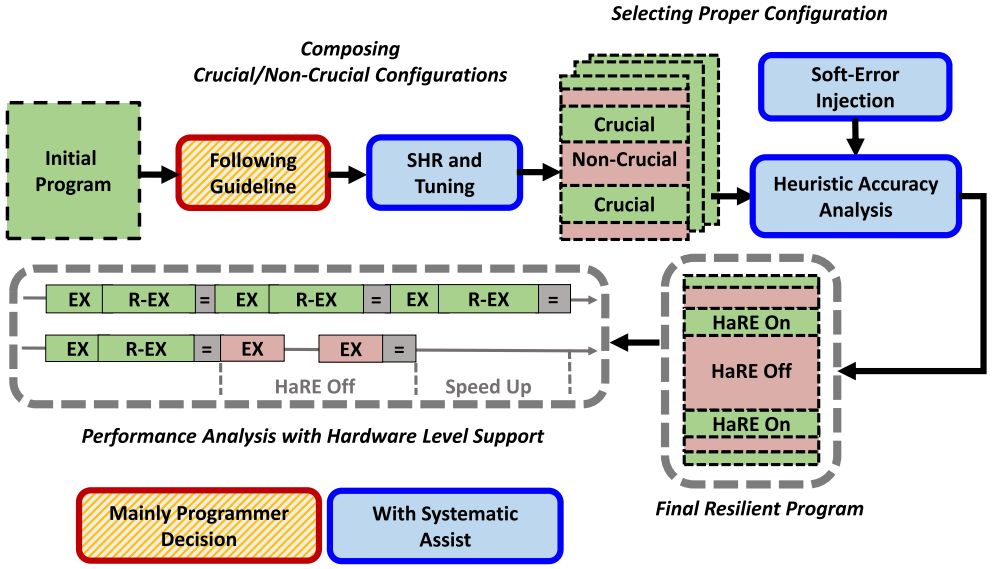

Fig. 3. Declarative resilience design flow.

and *loop-unrolling* are used to compose non-crucial instructions into code regions. These regions do not need to be all-correct at this point, and the programmer can identify as many regions as possible. Accuracy analysis is later performed to verify the accuracy loss of identified non-crucial code regions. It helps the programmer in selecting the proper combination of regions under certain constraints, such as soft-error rate and accuracy loss threshold. This combination is a subset of all the non-crucial regions, which is referred to as *configuration* in this article. With a proper configuration, the transformed program can be deployed on the proposed cross-layer architecture. In the current setup, programmer's effort is mainly spent in instruction classification. The steps involving *loop-unrolling* and *system-level bound checking* have been automated via systematic assist.

2.4.2 Fault Injection. After identifying crucial/non-crucial regions, these configurations are subjected to fault injection for evaluating and analyzing accuracy. Priors works [18, 24, 33] show that numerous novel methods have been devised to inject faults, which can be implemented in both hardware and software. The contrast between hardware and software methods lies mainly in the fault injection points they can access, the cost, and the level of perturbation. This article focuses on program-level fault injection, and thereby software-level fault injection mechanism is used to analyze program accuracy.

Soft errors can impact the data being processed with a noise phenomenon anytime, anywhere in the application. Due to the unpredictable and uncontrollable nature of soft-errors, random errors are introduced in non-crucial code regions (details in Section 4) via software-level fault injection. It is done in such a way that a random program variable prone to soft-errors (non-crucial region) is exposed to random values in the range determined by its data type. For example, consider a variable *var* of *integer* type in the non-crucial region. If *var* is subjected to fault injection, then any random value from the range of −2147483646 to +2147483647 (minimum and maximum values for an integer) would be committed to *var*.

In the fault injection analysis, a single error is injected in the non-crucial region of the program code. We apply both realistic and aggressive error rates to build up the confidence. The accuracy is

defined based on application dependent metrics (Section 3). The notion of the fault injection is that after applying the protection schemes, the resilience architecture ensures that store address and control flow instructions are always protected (using SHR or HaRE). The remaining vulnerable code includes data flow instructions, and the out of bound value is never committed. The accuracy loss of each region is obtained by applying fault injection to it. Multiple simulations (10 000 times in this article) are performed for each configuration of non-crucial regions to obtain the average program accuracy.

2.4.3 Heuristic Accuracy Analysis. Based on the programmer's decision, multiple combinations of crucial/non-crucial regions can be identified. The accuracy analysis requires an exhaustive search out of all these various combinations to find a near-optimal solution. However, to cover all the possible configurations, it needs to run 2^I setups (where I is the number of potential non-crucial regions), with each one simulated multiple times. For example, in the following sections we provide discussion over CNN-ALEXNET that contains $I = 9$ potential non-crucial regions. Analyzing CNN-ALEXNET requires running numerous setups ($2^9 = 512$) to find a near-optimal combination. This involves great computational effort when I is large like in the case of CNN-ALEXNET.

With the availability of multiple combinations, it is critical to reduce to a single near-optimal configuration that not only satisfies the accuracy threshold but also provides near-optimal performance. A heuristic has been devised that assists in converging to a single configuration out of all available crucial/non-crucial region combinations.

The goal of the heuristic is to maximize the number of non-crucial regions, such that the accuracy threshold constraint is also satisfied. Therefore, all regions are assumed non-crucial at the beginning of the analysis. Following this approach speeds up this process with fewer simulations to provide a near optimal configuration. If the program accuracy loss exceeds a programmer defined threshold, then a new configuration with one less non-crucial region is selected. The region with maximum accuracy loss in the previous configuration is considered as crucial. In case of multiple regions having similar accuracy loss, the one with minimum execution time is selected. This process is repeated until a crucial and non-crucial code region classification is achieved that satisfies the programmer's accuracy threshold.

To better explain this process, consider an example where an algorithm A has three regions R_1, R_2, and R_3, with x, y, z denoting individual accuracy losses, respectively, such that $x < y < z$. Initially, all three regions would be considered non-crucial to form a combination C_1. Suppose the output accuracy of C_1 is greater than the threshold. In this case, R_3 being the most sensitive region—R_3 has the highest accuracy loss (z)—would be dropped out of the configuration. A new combination C_2 would be formed, which contains R_1 and R_2 as non-crucial regions and R_3 as a crucial region. If the accuracy loss for the combination C_2 satisfies the accuracy threshold, then C_2 would be considered as the near-optimal configuration. Otherwise, R_2 would be dropped and a new combination would be formed. The process of analyzing the accuracy and reducing to a single configuration is a part of systematic assist, i.e., automated.

2.4.4 Resilience Switching. As shown in Figure 3, after selecting the proper configuration, the appropriate pragmas are placed in the program to switch between crucial and non-crucial. These are passed as HaRE on/off pragmas to the hardware. The hardware-software interface for declarative resilience is implemented using a special function instrumented in the programs. As mentioned earlier, bound checkers are important for inspecting the values to be committed to the memory. Bound checkers are nothing but control flow instructions to direct the program execution based on comparison results. Control flow instructions are required to be a part of crucial regions. Therefore, HaRE is turned "on" before the bound checker for protection against the soft-errors.

The final resilient program is executed on the proposed cross-layer architecture, with hardware level support implemented in a multicore simulator.

ALGORITHM 1: CNN Convolutional Layer Pseudo Code

```
1:  ConvolutionLayer(input, conv_out, tid, threads){
2:  for each tid do
3:      for each neuron do
4:          \* The following 3 level loop is Unrolled *\
5:          for (number of kernels k, kernel height, h, kernel width, w) do
6:              \* Assign temp_k/h/w *\
7:              HaRE Off
8:              conv_out += do_conv(input, temp_k/h/w)
9:              \* Update temp variables *\
10:             conv_out += do_conv(input, temp_k/h/w)
11:             \* Update temp variables *\
12:             :
13:             HaRE On
14:             \* Update k, h, w *\
15:         \* Bound_Checker not needed *\
16: }
```

2.5 Application Illustration

In this section, Convolutional Neural Network (CNN) is shown as an example to illustrate how to transform the program code in the proposed architecture. The application shown is handwritten digits recognition (MNIST) [40]. It mainly consists of four types of layers: input, convolution, fully connected, and an output layer.

The algorithm is inspired by animal visual cortex. It is widely used in image and video recognition. In CNN, individual neuron responds to a restricted region of visual space. The fields of different neurons partially overlap. The response of an individual neuron can be approximated by a convolution operation, which is also the major work in CNN. Because of its functionality, it is intuitive to consider the computation in convolution layer as non-crucial (as shown in Algorithm 1). Temporal variables are used in the iteration, to ensure that only temporal variables are updated in non-crucial regions. The loop is unrolled to get longer non-crucial instruction sequence. The computations (*do_conv*) are "inline" to avoid function call, since there should be no control-flow instructions in non-crucial regions. Note that compiler added spill code for a non-crucial region is local (temporal), and hence does not impact program correctness. Bound checkers need to be added in the iteration after HaRE is turned on, if necessary. In this case, no checker is needed (as shown in Algorithm 1 line 15), because the out of range value will be filtered out in the fully connected layer.

Similar flow is applied to the fully connected layer, as shown in Algorithm 2. Each neuron in the fully connected layer constitutes complete set of connections to all the ones in the previous layer. It does the computation based on the input from the previous layer and calculates a sigmoid for each neuron using respective weight values. The computations done are considered non-crucial. The variable *temp_O* is bounded from −10 to 90, since it gives an acceptable sigmoid range (0.00001 to 0.99999).

As described above, non-crucial code regions should only contain compute instructions, load instructions, and store instructions that access temporal variables. For example, the non-crucial

code region in the convolution layer of CNN-MNIST (shown in Algorithm 1) only contains following `x86 instructions: movl, addl, cltq, salq, addq, movq, subl, movslq, leaq, movsd, mulsd, addsd.`

ALGORITHM 2: CNN Fully Connected Layer Pseudo Code

```
1:  FullyConnectedLayer(input, fully_out, tid, threads) {
2:  for each tid do
3:      for each neuron do
4:          \* The following loop is Unrolled *\
5:          for each input i do
6:              HaRE Off
7:              temp_O+=(input(temp_i) * weight(temp_i))
8:              temp_i = 1
9:              temp_O+=(input(temp_i) * weight(temp_i))
10:             :
11:             temp_i = 2, 3, ....
12:             temp_O+=(input(temp_i) * weight(temp_i))
13:             HaRE On
14:             \* Update i *\
15:             Bound_Checker(temp_O)
16:         fully_out = Sigmoid(temp_O)
17: }
```

3 NON-CRUCIAL REGIONS OF TARGET APPLICATIONS

Machine-learning and graph analytic applications are ubiquitously used in many domains where systems could be exposed to soft-errors, such as high altitude or temperature environments. Due to their unique structure and computational behaviors, research has been done on relaxing their accuracy for performance benefits. Likewise, these applications have potential to provide performance improvements using the proposed architecture with resilience–accuracy trade-off. We evaluate six machine-learning and nine graph analytic applications in this article.

3.1 Machine Learning

Machine-learning algorithms work on massive data and perform perception computations. These workloads are used in numerous applications, such as image recognition, video analysis, and natural language processing. Such applications have the potential to be deployed in safety-critical systems where they face resilience challenges [58]. On the contrary, due to their inherent heuristic nature, individual floating-point calculations in machine-learning applications hardly impact program outcome. In this article, we evaluate four multi-threaded versions of the most commonly used convolutional neural networks: AlexNet (ALEXNET) [25], VGG [62], handwritten digits recognition (MNIST) [40], and recognition of German traffic signs (GTSRB) [54]. They all mainly consist of four types of layers: input, convolutional, fully connected, and output layers. The rate of "correct classification/the number of tests" is defined as the accuracy of an application. For example, when applying 100 handwritten digits through CNN-MNIST, if 95 are classified correctly its accuracy is defined as 95%. The accuracy loss is normalized to the program accuracy in soft-error free condition. The benchmark setups are shown in Table 3.

For the neural networks discussed in this article, most part of their execution time is spent in convolution and fully connected layers. The computations within these layers are considered non-crucial. In K-Nearest Neighbors (KNN), objects are classified using several known examples. Class of the new object is determined by a majority vote over its neighbors. The computation of the distances between the new object and the known object is also considered non-crucial. This is because each example has minimum impact to the program. Moreover, the distance calculation of one neighbor is independent of the other neighbor, and thereby perturbations due to soft-errors do not propagate.

Theoretically, the input layer of these benchmarks can also be considered non-crucial. In the case of image recognition applications (such as CNN-MNIST), few perturbed pixel values can hardly affect the program outcome. Moreover, it is highly possible that they will be masked in convolution computations. However, in our benchmark setup, input layers load files using I/O system calls, which are not in the context of this article. Thus, input layers are always considered crucial. Alternatively, output layer is highly sensitive to errors/faults, since it is responsible for predicting the output classes. Therefore, it is preferable to consider output layer as crucial.

3.2 Graph Analytics

Graph analytic benchmarks traverse the vertices in the input graph and compute based on the connectivity and weights on the connected edges. They may consist of multiple phases, which may serve different functionalities including synchronization of threads. Most of these computations within each phase can be identified as non-crucial. Due to their parallel strategy, the benchmarks may iterate over the input graph multiple times. Thus, certain iterations can also be considered non-crucial. In this article, we evaluate nine multi-threaded graph analytic benchmarks from CRONO suite [1]. Like machine-learning applications, all the graph analytic workloads are shown in Table 3. The accuracy metrics are defined based on individual benchmark's outcome.

Here, we use Triangle Counting (TRI-CNT) as an example. It measures vertex connections, which is used in applications such as web connectivity. This benchmark consists of three phases. The first phase adds side counts from each edge to a global data structure for each vertex. The second phase reduces these added counts. It involves more computations, specifically divisions to get triangles per vertex. These triangles are accumulated per vertex into a total triangle count in each thread. Bound checkers are used to ensure the total triangle count are less than the total edge count. The final phase gets the total number of triangles and is sequential involving only the master thread to sum up the total number of triangles. All the computations done within these phases can be considered non-crucial.

The non-crucial regions of all the graph analytic benchmarks other than Single Source Shortest Path (SSSP) are classified in an equivalent manner as Triangle Counting. SSSP goes through the input graph multiple times and updates the result matrix when smaller distance values are found. If all the iterations are considered non-crucial, then the accuracy loss of SSSP turns out to be more than 99%. The reason for such a high accuracy loss is because of the propagation of the perturbed values. An experiment was performed to observe the effects of protecting last 20% iterations against protecting none. We observed a lower accuracy loss as compared to the naive approach and concluded that wrong values due to soft-errors are highly likely to be masked in later iterations. Thus, we define the first 80% iterations as non-crucial. It is to be noted that, in the real case, the programmer would define how many iterations need to be crucial based on the accuracy analysis.

For most of the graph analytic benchmarks, accuracy is defined as the percentage difference of the result value (by comparing the perturbed and the golden outputs) of each vertex, such as PageRank, SSSP, and so on. For searching algorithm benchmarks DFS and BFS, the accuracy is

defined as the percentage of correctly checked vertices. For Triangle Counting, it is defined as the percentage difference in the number of counted triangles for the perturbed output and the error-free output.

3.3 Accuracy Threshold Selection

The heuristic accuracy analysis requires the programmer/user to provide an accuracy threshold to mark regions as crucial or non-crucial. The higher the accuracy threshold, the higher would be the room for letting accuracy drop. Accuracy threshold selection depends on various aspects, such as the type of system and application. An accuracy threshold selected for a safety/life critical system would be comparatively lower as compared to the case where a system must guarantee response within specified time constraint. Similarly, accuracy threshold can vary depending on the sensitivity of applications and the environment where they are deployed. In the context of machine-learning applications, the accuracy threshold could be decided based on various parameters, such as *frames per second (fps)*. Consider an example of advanced driver-assistance systems (ADAS). Convolutional neural networks (CNNs) are deployed in such *sensor-to-decision* systems to process images and predict prospective actions. A stream of images is provided to the CNN with a certain *fps* rate to predict the next action. Usually, the *fps* rates range from 30 to 60 frames per second [6]. The higher the fps rate, the more accurate and reliable decision would be made by the system. Depending on conditions around the vehicle, the *fps* value could be allowed to change. A high *fps* rate would be required in case of heavy traffic, and lower *fps* rates would be acceptable if the traffic is smooth and light. For the latter case, lowering the *fps* rate and losing some frames—say, 5% of the total 60 image frames, i.e., ∼3 images—from the image stream would not impact the final decision significantly. Therefore, in this case an accuracy threshold of 5% would be acceptable.

Graph applications are tolerant to random errors, as long as the nodes with higher connectivity are not affected [9]. In the recent work [42], it has been shown that even at aggressive error rates graph algorithms show highly accurate results. Therefore, there is a negligible probability that a single injected error would affect nodes with high connectivity and would surely not impact the final response significantly. In terms of graph analytic workloads, accuracy threshold selection depends on graph inputs provided to the workload due to input dependence. For sparse graph input there is room for selecting a higher accuracy threshold. However, dense graph inputs require relatively lower accuracy threshold constraints due to higher number of connections per vertex.

4 METHODOLOGY

4.1 Performance/Energy Analysis Setup

We use the Graphite multicore simulator [36] to model the proposed cross-layer resilient architecture. The default architecture parameters are summarized in Table 2. We model all hardware resiliency mechanisms proposed in Reference [57]. The hardware-software interface is implemented using a special function instrumented in the application, which passes the HaRE on/off pragma to the simulator. With the in-order single-issue core setup, we assume a five-stage pipeline. The HaRE "on" switch needs three cycles to create a safe state and start capturing signatures. For HaRE "off" switch, the pipeline needs to be flushed with an additional one cycle delay to re-execute and check the previous instruction sequence. Redundant store address calculation (SHR) incurs one cycle delay, since it needs to stall the compute pipeline. The completion time is broken into following categories:

(1) *Instructions:* Time spent retiring instructions.
(2) *L1-I Fetch Stalls:* Stall time due to instruction cache misses.

Table 2. Architectural Parameters

Architectural Parameter	Value
Core(s)	64—In-Order
Memory Subsystem	
L1-I/D Private Caches (per Core)	32KB, 4-way Set Assoc., 1 cycle latency
L2 Shared Cache (per Core)	256KB, 8-way Set Assoc., 8 cycle latency, Inclusive
Coherence Protocol	Directory, Invalidation-based, MESI
DRAM Memory Interface	8 controllers, 5GBps/controller, 100ns latency
Electrical 2D Mesh with XY Routing	
Hop Latency	2 cycles (1-router, 1-link) + link contention
Flit Width	64 bits

(3) *Compute Stalls:* Stall time due to waiting for functional unit (ALU, FPU, Multiplier, etc.) results.
(4) *Memory Stalls:* Stall time due to load/store queue capacity limits, fences and waiting for load completion.
(5) *Branch Speculation:* Stall time due to mispredicted branch instructions.
(6) *Synchronization:* Stall time due to waiting on locks, barriers, and condition variables.
(7) *Resilience:* Stall time due to hardware level resilience schemes.

We evaluate dynamic energy in this article. For energy evaluations of on-chip electrical network routers and links, we use the DSENT [65] tool. Energy estimates for the L1-I, L1-D, and L2 (with integrated directory) caches are obtained using McPAT [29]. The evaluation is performed at the 11*nm* technology node to account for future technology trends.

4.2 Accuracy Analysis Setup

Based on the guidelines for classifying non-crucial regions, a programmer can form multiple combinations of crucial/non-crucial regions—each combination is referred to as a *configuration*. Each configuration contains non-crucial code regions, which consist of contiguous compute instructions, load instructions, and store instructions that access temporal variables. The heuristic accuracy analysis (cf. Section 2.4) returns a near optimal configuration out of multiple combinations that not only satisfies the accuracy threshold but also provides better performance. Keeping in mind the effects of soft-error strikes in non-crucial regions, we believe injecting error(s) to the variable(s) before committing to global is sufficient. Considering soft-errors as bit-flips, which can happen anywhere in the non-crucial region and have unpredictable effects to the variables about to commit, we mimic them using random values. The injected random value ranges are determined based on the data types of the variables (cf. Section 2.4.2). For example, if the data type is *double*, random values ranging from 1.7E +/− 308 are injected. Using such a metric of injecting faults based on a range allows us to cater for nearly all the possible bit-flip scenarios. In this article, the results are shown for the single error injection analysis. The impact of applying aggressive error rates on the accuracy of each benchmark is also discussed.

4.2.1 Soft Error Rates (SERs). In reality, the occurrence of soft-errors is rare [21], therefore, the primary focus of this article is on a single soft-error strike happening during the program's execution. However, in extreme cases such as harsh environmental conditions in the terrestrial environment, the occurrence of soft-errors could be high [19]. To explore the accuracy tradeoff and evaluate soft-error effects in such extreme cases, aggressive error rates are also applied. The error rate is applied as the probability of single iteration being perturbed. For example, suppose there are total 1000 loop iterations in the non-crucial region. When the error rate of 1% is applied, on average there would be 10 iterations (1% of total iterations) with random values injected in them via software level fault injection. In case there are both crucial and non-crucial instructions in the iterations, errors are injected with a probabilistic distribution based on the execution time of the non-crucial instructions. Building on to the aforementioned example, suppose within each iteration of the non-crucial loop nearly 70% of the total time is spent in executing non-crucial instructions. In such a case, on average 7 iterations (70% of 1% of total iterations) would be subjected to fault injection.

4.3 Hardware Overhead & Code Footprint

The data checker (SHR) is implemented in the application using regular instructions, so it adds to the instruction footprint but incurs no additional hardware overhead. For performance purposes, we unroll the program loops to create contiguous non-crucial regions that introduces code size overheads. This highly depends on individual program and compiler setup. For the benchmarks evaluated in this article, code footprint is not a major concern. Therefore, it is not quantitatively analyzed in this article.

The proposed scheme introduces similar hardware overhead as suggested by HaRE [57], including the speculative store buffer (SSB), duplicated register files, and the hardware signatures. It uses existing HaRE hardware to perform redundant store address checking. Overall, the storage overhead is ~5K bits per core.

4.4 Benchmark & System Setups

The benchmark setups are shown in Table 3. We evaluate nine graph analytic benchmarks from CRONO suite [1], with different graph inputs to evaluate input dependency. California road network [27] is used as a sparse input and mouse brain graph [31] as a dense input. For MNIST machine-learning benchmarks (CNN, MLP, KNN) handwritten digit dataset [26] is used. While CNN-GTSRB uses German traffic sign [63] as an input. ImageNet [11] is provided as an input to CNN-ALEXNET and CNN-VGG.

The following system setups are used for evaluation.

(1) **BASELINE** is the original program without any resilience schemes.
(2) **HaRE** redundantly executes all program instructions.
(3) **DR** is the proposed cross-layer architecture based on declarative resilience architecture, which selectively applies HaRE and SHR for crucial and non-crucial code regions, respectively.

5 EVALUATION

5.1 Non-Crucial Region Selection

The selection of code regions as non-crucial is considered based on the heuristic described in Section 2.4. Consider the example of CNN-ALEXNET to illustrate this selection process. As shown in Table 4, all convolution (C1–C5), and fully connected (F1–F3) layers in CNN-ALEXNET have relatively low accuracy loss. When single error is injected in these layers, the maximum accuracy

Table 3. Benchmark Setup

Application	Setups
Machine Learning	
CNN-VGG (IMAGENET [11])	16 convolutional, 3 fully connected layers
CNN-ALEXNET (IMAGENET [11])	5 convolutional, 3 fully connected layers
CNN-GTSRB (GTSRB [63])	2 convolutional, 2 fully connected layers
CNN-MNIST (MNIST [26])	1 convolutional, 1 fully connected layer
MLP-MNIST([26])	2 intermediate layers
KNN-MNIST([26])	5 nearest neighbors
Graph Analytic	
SSSP, D-STAR, BFS PAGERANK, TRI-CNT, DFS, CON-COMP, COMM, BTW-CENT	California Road Network (Sparse Graph Input), Mouse Brain Retina 3 (Dense Graph Input)

Table 4. Program Accuracy of CNN-ALEXNET When Single Error is Injected in the Each Region (Over 10,000 Runs)

Benchmark	Layers/Regions	Accuracy Loss (%)
CNN-ALEXNET	C1	0.3
	C2	0.4
	C3	1.9
	C4	0.4
	C5	0.3
	F1	0.8
	F2	1.0
	F3	1.3
	Output	13.1

C, convolutional layer; F, fully connected layer; Output, output layer.

loss of 1.9% is observed. In contrast, the output layer observes a loss of more than 10%. The programmer sets an accuracy threshold for the selection heuristic to mark certain code regions as crucial. We set the accuracy loss threshold at 2% (for both machine-learning and graph analytic workloads), which classifies the output layer as crucial and all convolution and fully connected layers as non-crucial.

To explore the accuracy trade-off of selected non-crucial regions, aggressive error rates were also considered. However, we show the results for single error injection in our performance analysis. Overall, DR is able to select configurations with reasonable amount of non-crucial code, resulting in low accuracy loss even with high error rates. The accuracy loss in different benchmarks highly depends on the code structure and the functionality. For code regions with similar functionality in a benchmark, the more execution time the code region has, the more accuracy loss it may contribute. This behavior is observed in the convolution layers of CNN. However, this pretext would not hold if the code regions have different functionalities, such as the output layer in all machine-learning

Table 5. Selected Configurations of Machine-learning Benchmarks When Single Error Is Injected in the Program (Over 10,000 Runs)

Applications	Selected Non-Crucial Regions	Non-Crucial Time (%)	Accuracy (%)
CNN-ALEXNET	C: 1–5 + F: 1–3	91	99
CNN-VGG	C: 1–16 + F: 1–3	92.5	98.8
CNN-GTSRB	C: 1, 2 + F: 1, 2	88	99.1
CNN-MNIST	C + F	87	99.3
MLP-MNIST	I: 1, 2	75	99.91
KNN-MNIST	Distance	94	99.9

C, convolutional layer; F, fully connected layer; I, intermediate layer.

Table 6. Selected Configurations of Graph Benchmarks When Single Error is Injected in the Program (Over 10,000 runs)

Applications	Selected Non-Crucial Regions (Heuristic Analysis)	Non-Crucial Time Time (%)		Accuracy(%)	
		Sparse	Dense	Sparse	Dense
SSSP	80% Iterations	58	64	98.2	98.91
D-STAR	Distance	72	91	99.7	99.45
PageRank	Ranking	74.3	78	99.9	99.3
TRI_CNT	AddEdges, Redu, GetTri	89	86	99.4	99.05
BFS	Vertex Check	21.4	27.8	99.97	99.7
DFS	Vertex Check	77	81	99.9	99.6
COMM	Mod, Recons, Reduce	88	89	99.76	99.29
CON-COMP	Denoting	58	74	98.87	98.38
BTW-CENT	Distance, Centr	84	86	99.86	99.34

The Non-crucial region names represent their functionalities as in CRONO [1].

benchmarks. It does not have much execution time, yet it incurs relatively higher accuracy loss. This is because errors in the output layer could directly affect the final program outcome.

Tables 5 and 6 show the percentage time spent in non-crucial code and the percentage accuracy loss of the selected configurations at a single injected error per program execution. All machine-learning benchmarks consider their output layers as crucial. The second column in Table 5 represents which layers of the benchmark were considered as non-crucial. For example, "C: 1–5 + F: 1–3" for CNN-ALEXNET depicts that all convolution layers from 1 to 5 were selected along with all the fully connected layers from 1 to 3. Moreover, the selected non-crucial code regions account for >75% of the execution time, while the accuracy loss is observed at less than 1% (satisfying the threshold).

For the graph benchmarks, when different input graphs are applied the observed accuracy loss is relatively stable, while the time spent in the selected non-crucial code regions vary significantly. For example, for the path finding heuristic D-STAR, edges are involved in the distance calculation. A sparse graph has less edges per vertex, while a dense graph has much higher edge connectivity. Therefore, the work done in non-crucial region for a dense graph is much larger for a dense graph as compared to a sparse graph. However, the accuracy loss of the selected non-crucial code region remains under 1% for both input graphs. Similar behavior is observed for other graph

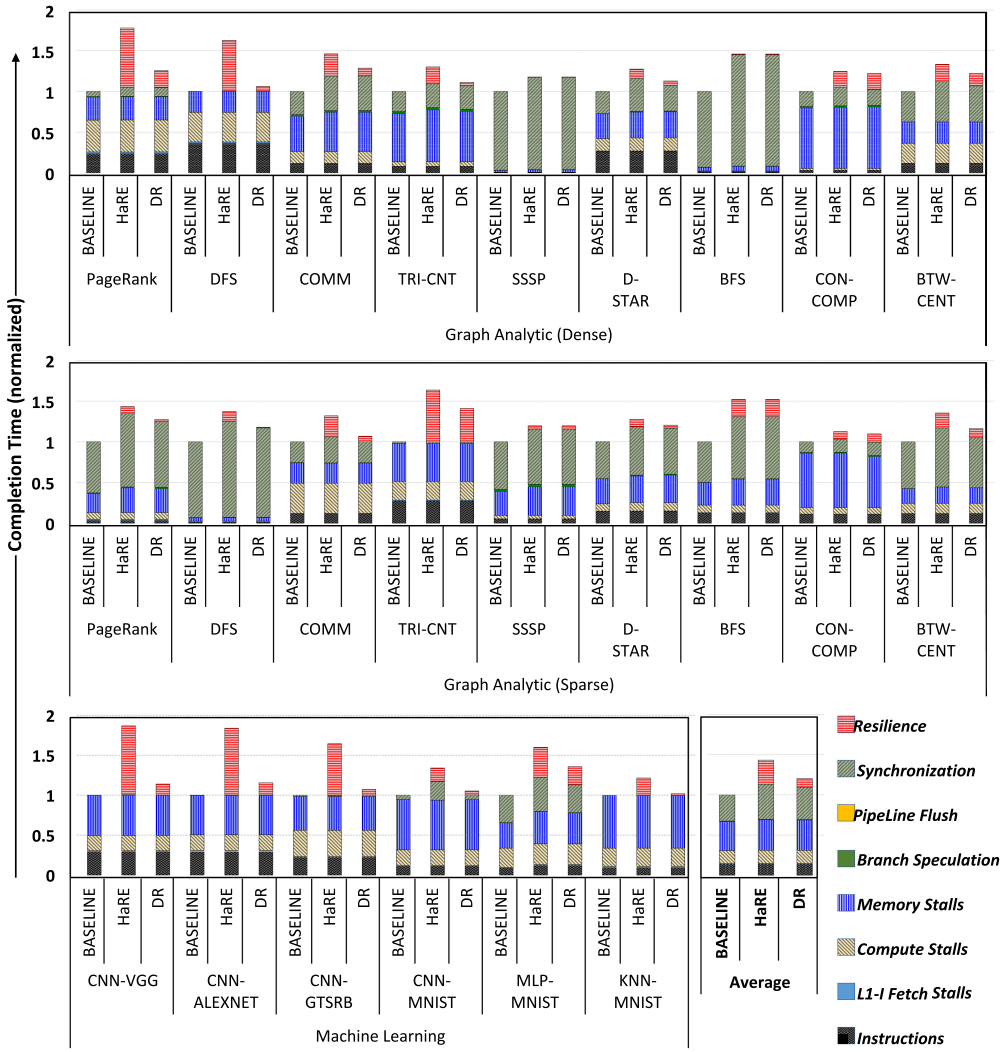

Fig. 4. Completion times of workloads with selected configurations using proposed cross-layer architecture.

benchmarks, where the worst-case accuracy loss is observed at 1.8% among all graph benchmarks. The time spent in non-crucial code regions is directly proportional to the expected performance gains from executing the graph benchmarks under the proposed DR architecture. We evaluate this performance behavior in the next section.

5.2 Performance/Energy

Under normal conditions, the probability of soft-error strike is very low (less than 1 per day for the current technology node [59]). To reveal the soft-error effects on program execution, a single error per program execution is used. The accuracy threshold is set to 98% to select the proper configurations. The completion time of selected configurations of each application is plotted in Figure 4.

The BASELINE completion time for most benchmarks is dominated by memory stall and/or synchronization. This is due to benchmarks' inherent characteristics: numerous memory accesses to shared data, and heavy contentions on locks or barriers. For graph analytic workloads, such as SSSP and BFS, results show higher synchronization delay when using the dense graph input. This is because of higher number of shared edges, which causes more lock contention. For benchmarks like PageRank and DFS, more time is spent in synchronization for the sparse graph input. This is because they use coarse-grain locks over the vertices (not the edges).

HaRE performs reasonably well for most benchmarks, because it performs local redundant execution and exploit core-level locality. Moreover, it hides re-execution latency behind cache miss stalls. Memory stalls of HaRE are increased over BASELINE, because memory operations trigger re-executions (such as invalidating a cache line). Messages need to wait until the re-execution completes. The synchronization of HaRE also increases over BASELINE due to the re-execution time of instructions within locks or barriers.

When applying DR, all machine-learning applications show remarkable performance improvement over HaRE. DR reduces the completion time of CNN significantly compared to HaRE (from 1.83× to 1.15× for ALEXNET). This is because HaRE is not able to hide resilience overheads, meanwhile a major amount of computations are identified as non-crucial in DR. For benchmarks with high resilience overhead in HaRE, DR is always able to reduce it. Note that the time spent in non-crucial regions (shown in Tables 6 and 5) is not directly proportional to the performance gain in the proposed architecture across benchmarks. This is mainly due to the following reasons. (1) The non-crucial time reported in Tables 6 and 5 refers to the execution time of the whole layer/phase, which includes locks and checkers. Thus, the execution time of actual non-crucial instructions could be much smaller. (2) For benchmarks having highly contended fine-grain locks, such as SSSP and BFS, their resilience overhead is hidden behind synchronization delay. Although DR reduces the re-execution within locks, they may not overcome the delay added due to software level checker, and on/off switching of HaRE. (3) For benchmarks with coarse-grain locks, such as DFS, DR is able to reduce the synchronization delay (sparse input) because of the preferable longer instruction sequence within locks. In the case of COMM, DR causes workload imbalance between threads and slightly increases the synchronization delay of barriers. We observe that SSSP and BFS do not provide much performance benefit. This is mainly due to the fact that most of the computations are done within the locks, which is why we observe high synchronization in the reported results. The computations done within the locks can be considered non-crucial. However, synchronization instructions such as barriers and locks need to be protected. This limits us from unrolling the non-crucial instructions and thus, we do not observe much benefits from the proposed architecture. In case of CON-COMP, we do not observe performance gains over HaRE because of the structure of the algorithm. This algorithm contains many indirect accesses of form $A[A[i]]$, which need to be considered as crucial. Protection of such instructions leaves very few non-crucial instructions, which cannot be unrolled for performance benefits. Overall, the proposed DR architecture shows significant performance improvement over HaRE. It reduces the performance overhead of resiliency from ~1.43× to ~1.2× on average.

As shown in Figure 5, in general the dynamic energy results follow the same trend as performance results. The energy overhead mainly comes from the additional computation, memory access, and network traffic, generated by redundant execution. Although HaRE re-executes all instructions, its energy overhead is much less than 2X (~1.56× on average). The energy improvement is due to per-core re-execution exploiting core-level locality. Additionally, the costly cache misses are not duplicated. DR reduces energy overhead significantly (~1.32× on average) due to the fact that fewer instructions are re-executed.

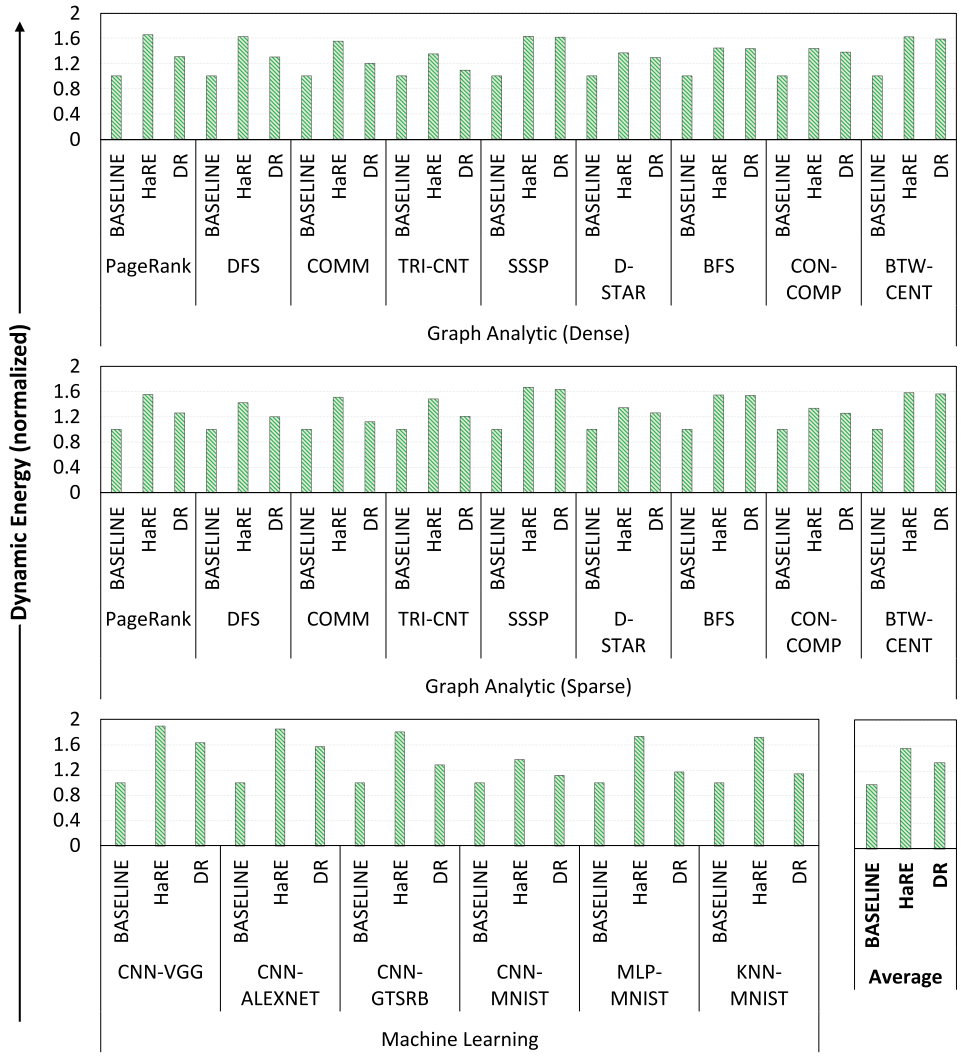

Fig. 5. Normalized energy numbers of selected configurations using proposed cross-layer architecture.

5.3 Configuration Selection

Soft-error rates change due to different conditions in real world environmental conditions. In addition, the acceptable accuracy loss is determined from case to case. We introduce heuristic accuracy analysis (Section 2.4) in this context to help the programmer select the proper configurations. Figure 6 shows the performance improvements over HaRE of selected configurations, at different accuracy thresholds with different error rates. Some benchmarks, such as D-STAR, have relatively low accuracy loss (<=1% for D-STAR) in the original configurations, which can meet the lowest accuracy threshold (3%). Their configurations are not changed when applied with different error rates and accuracy thresholds. Hence, their performance numbers remain constant in this analysis, and are not shown in the figure.

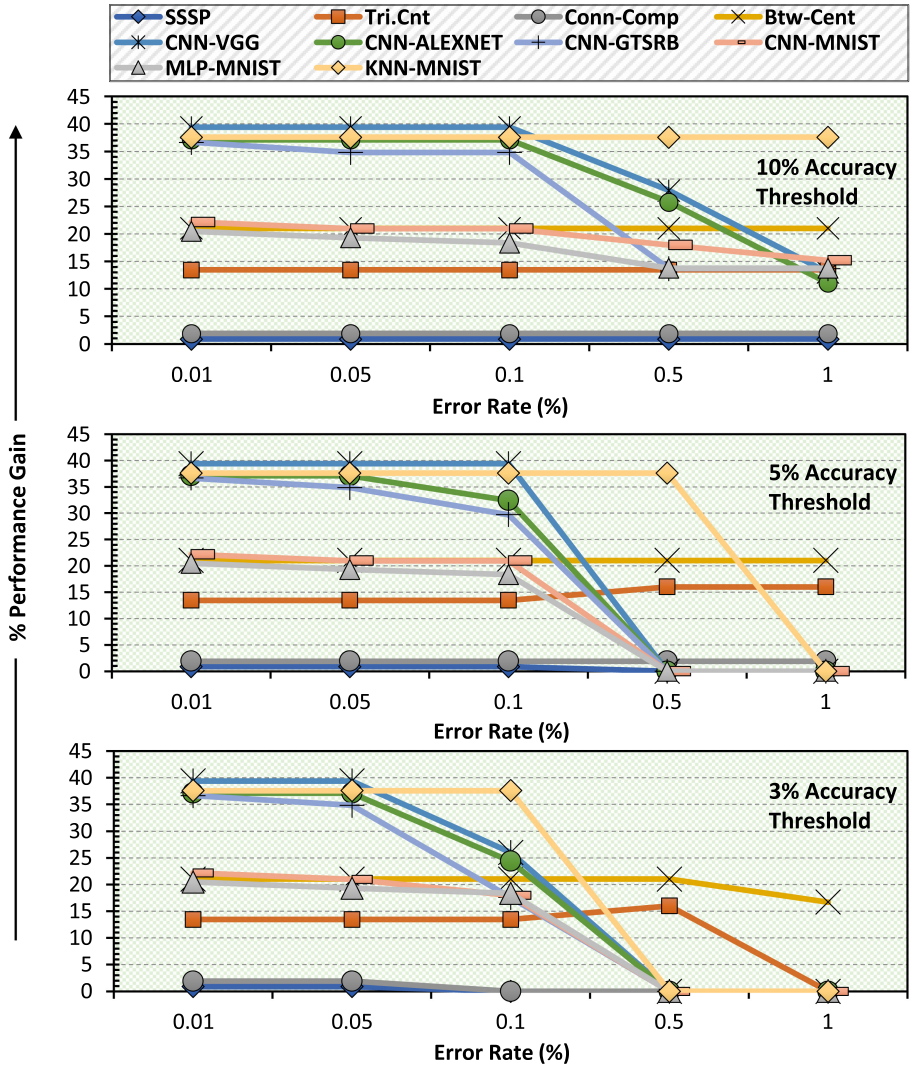

Fig. 6. Performance improvements over HaRE at 10%, 5%, and 3% accuracy thresholds with various error rates. Significant improvement numbers are observed at higher accuracy thresholds.

As shown in Figure 6, three different accuracy thresholds are applied, from top to bottom, 10%, 5%, and 3% accordingly. In general, a higher accuracy threshold and lower error rate results in configurations with more non-crucial code regions. For example, when using an accuracy threshold of 10%, AlexNet is able to get the best performance at 0.1% or lower error rates. According to Table 5, all of its convolution and fully connected layers are considered as non-crucial at this point, which contribute 91% of the program's execution time. Thus, the proposed cross-layer architecture provides exceptional performance improvement over HaRE (37%). As the error rate increases, the same configuration cannot satisfy the 10% accuracy threshold. It gets to 31% at an error rate of 0.5%, which was 37% earlier (error rate of 0.1%). Using the proposed heuristic accuracy analysis (Section 2.4), we are able to get the accuracy loss back by making the fifth convolution layer and

the third fully connected layer crucial. However, this results in less non-crucial code (76% of execution time), thus has less performance improvement (26%). In extreme cases, the performance improvement can reduce to 0%, which points to the fact that all code regions would have to be considered as crucial, and effectively employ only HaRE for protection.

6 RELATED WORK

6.1 Resilience Scheme for Multicore

Symptom-based mechanisms [52, 70] have low coverage, since they rely on coarse-grain detectors, such as fatal-traps, hangs, panics, and so on. However, they incur low area, power, and performance overheads. *Software* solutions such as instruction duplication [14, 41, 46], and invariant checking [3, 34, 44] improve coverage, but these solutions incur higher overheads. To deliver holistic coverage, several proposals utilize temporal and spatial redundant execution. PROFiT [47] introduces software control in fault tolerance and allows users to fine-tune the tradeoff between performance and reliability. Redundant multithreading [38, 43, 50, 69] uses the processor's multithreading (within same or different cores) contexts to run two copies of the same thread, where the trailing thread verifies the results of the leading thread. This approach is generalized as *n-modular redundancy* [4, 61], where n copies of the same thread are executed and verified in parallel. These techniques incur significant performance and energy overheads, because multithreaded applications are unable to exploit the hardware's thread-level parallelism. To improve performance, researchers have explored selective resilience within applications [30, 45]. These schemes obtain efficiency by providing high resiliency for high vulnerability code; however, they tradeoff performance with soft-error coverage. Similarly, selective SWIFT-R [48] uses the concept of selective hardening to design reduced overhead and flexible mitigation techniques by trading off reliability (coverage) and resilience overheads. Relax [10] introduces a framework to detect soft-error at hardware level and recover them at software level. However, since it allows the potentially faulty states to be committed, its coverage is compromised. The proposed paper is focused more toward trading off resilience overheads with program accuracy while keeping the soft-error coverage unaffected. Prior work [8] applies selective redundancy to register duplication in software-based techniques and analyzes its efficiency for microprocessors. However, the proposed declarative resilience framework not only focuses on registers but applies re-execution at a per-core granularity and implements resilience cache coherence protocol.

Prior work [56] also focuses on applying the strategy of selective resilience. However, this article extends the work presented in Reference [56] in terms of various aspects. A simple and structured crucial/non-crucial code region classification has been presented. Moreover, a rigorous accuracy analysis has been proposed that uses program level fault injection to determine the impact of soft-errors on the classified regions. Last, in the prior work [56], only four different PARSEC [5] applications were targeted via selective resilience. This article focuses more on instrumenting ubiquitously used real-time machine-learning and graph analytic applications with the declarative resilience framework.

6.2 Approximate Computing

Research has explored approximate programming [12, 13, 37, 60], which relaxes program accuracy when possible. Esmaeilzadeh et al. [12] focus on developing a language to generate code that executes on approximate hardware. Venkatagiri et al. introduce a framework [67] that quantifies the quality impact of perturbations in all dynamic instructions in an execution with a high accuracy. Moreover, it estimates the approximation capability of general programs and applications. Alongside these techniques, approximate kernels [51], and approximate cache

architectures [32, 35] have been developed for performance benefits. The proposed architecture is different from these works mainly in the following two aspects. First, our approach is not limited to approximate hardware and is more general, i.e., find code regions of a program that can be potentially "approximated" without significantly impacting program outcome. Unlike approximate computing schemes, the proposed architecture does not actively relax accuracy. Program accuracy may be compromised only when soft-errors perturb non-crucial code. Second, soft-errors introduce new challenges, since code in the non-crucial regions can unexpectedly affect other parts of the program and compromise program correctness thus, identifications for approximate computing cannot be directly applied to declarative resilience. However, the proposed architecture can benefit from profiling schemes of approximate computing, which can assist programmers to identify potential crucial/non-crucial code regions.

6.3 Crucial/Non-Crucial Code Identification

Works have been done on selecting crucial/non-crucial code of a program using different criteria, such as Rely [7], which is a programming language that enables developers to specify the reliability requirements for program functions. They verify the program reliability with respect to the specification of the underlying hardware system. Similarly, a program reliability optimization algorithm is proposed in Reference [55] that selects code snippets for program level protection considering their reliability-wise lower or higher impact on the program output. In terms of code region selection, the proposed declarative resilience framework is mainly different from previous schemes in the following ways. (1) In declarative resilience, programmers are involved in identifying the code regions, however, they cannot actively reduce reliability. If it is possible for some code to have critical impact on program outcome, then it cannot be marked as non-crucial. (2) Research (such as SoftBeam [16]) points out that different micro-architectures as well as different instructions have different inherent soft error vulnerability. However, in declarative resilience, we do not classify code according to its hardware level vulnerability. This is based on the insight that vulnerability cannot be used directly to guide the code's impact on the program. High vulnerability code may not have severe effects. However, soft errors in low vulnerability code may still crash the program, although the possibility of the soft error happening is low.

6.4 Algorithm Level Accuracy Tradeoff

Prior work has been done on fast algorithms with lower accuracy [15, 20, 39]. These algorithm level schemes have better performance; however, they do not provide protections to soft-errors. Our framework can be applied on top of such schemes seamlessly for resilience purposes. Due to their inherent accuracy relaxation, the proposed framework may bring less impact to the program outcome. Algorithms such as D* [64] approximate path planning and provide better performance than exact algorithms with relaxed accuracy. Such algorithms open up an opportunity to apply *declarative resilience* if program accuracy can be relaxed actively.

7 CONCLUSION

This article proposes a novel *declarative resilience* architecture for graph analytic and machine-learning applications. The key idea is to explore resilience overhead tradeoff with program accuracy, while not compromising soft-error coverage and safe execution of the program. We demonstrate a design flow to transform a program onto the architecture. It guarantees program correctness and incurs an average ~1.2× performance overhead over a system without resilience. This is an average 16% performance improvement over state-of-the-art hardware resilience scheme (HaRE) that protects the whole program. Furthermore, the proposed declarative resilience architecture reduces energy overheads by ~15% over HaRE.

REFERENCES

[1] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. 2015. CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'15)*. 44–55. DOI: http://dx.doi.org/10.1109/IISWC.2015.11

[2] Konstantinos Aisopos and Li-Shiuan Peh. 2011. A systematic methodology to develop resilient cache coherence protocols. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM, New York, NY, 47–58. DOI: http://dx.doi.org/10.1145/2155620.2155627

[3] T. M. Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO'99)*. 196–207. DOI: http://dx.doi.org/10.1109/MICRO.1999.809458

[4] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. 2005. NonStop reg; advanced architecture. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'05)*. 12–21. DOI: http://dx.doi.org/10.1109/DSN.2005.70

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 72–81.

[6] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to end learning for self-driving cars. *CoRR* abs/1604.07316 (2016). Retrieved from http://arxiv.org/abs/1604.07316.

[7] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*. ACM, New York, NY, 33–52. DOI: http://dx.doi.org/10.1145/2509136.2509546

[8] E. Chielle, J. R. Azambuja, R. S. Barth, F. Almeida, and F. L. Kastensmidt. 2013. Evaluating selective redundancy in data-flow software-based techniques. *IEEE Trans. Nuclear Sci.* 60, 4 (Aug. 2013), 2768–2775. DOI: http://dx.doi.org/10.1109/TNS.2013.2266917

[9] Paolo Crucitti, Vito Latora, Massimo Marchiori, and Andrea Rapisarda. 2003. Efficiency of scale-free networks: Error and attack tolerance. *Physica A: Stat. Mech. Appl.* 320 (2003), 622–642.

[10] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 497–508. DOI: http://dx.doi.org/10.1145/1815961.1816026

[11] J. Deng, W. Dong, R. Socher, L. J. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*. 248–255. DOI: http://dx.doi.org/10.1109/CVPR.2009.5206848

[12] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture support for disciplined approximate programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM, New York, NY, 301–312. DOI: http://dx.doi.org/10.1145/2150976.2151008

[13] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, Washington, DC, 449–460. DOI: http://dx.doi.org/10.1109/MICRO.2012.48

[14] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic soft error reliability on the cheap. *SIGPLAN Not.* 45, 3 (Mar. 2010), 385–396. DOI: http://dx.doi.org/10.1145/1735971.1736063

[15] Yangguang Fu, Mingyue Ding, and Chengping Zhou. 2012. Phase angle-encoded and quantum-behaved particle swarm optimization applied to three-dimensional route planning for UAV. *IEEE Trans. Syst., Man Cyberneti. Part A: Syst. Hum.* 42 (2012), 511–526.

[16] M. Gschwind, V. Salapura, C. Trammell, and S. A. McKee. 2011. SoftBeam: Precise tracking of transient faults and vulnerability analysis at processor design time. In *Proceedings of the IEEE 29th International Conference on Computer Design (ICCD'11)*. 404–410. DOI: http://dx.doi.org/10.1109/ICCD.2011.6081430

[17] S. K. S. Hari, S. V. Adve, and H. Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*. 1–12. DOI: http://dx.doi.org/10.1109/DSN.2012.6263960

[18] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (Apr. 1997), 75–82. DOI: http://dx.doi.org/10.1109/2.585157

[19] Texas Instruments. 2016. Texas instruments soft error FAQs. Retrieved from http://www.ti.com/support-quality/faqs/soft-error-rate-faqs.html.

[20] G. R. Jagadeesh, T. Srikanthan, and K. H. Quek. 2002. Heuristic techniques for accelerating hierarchical routing on road networks. *IEEE Trans. Intell. Transportat. Syst.* 3, 4 (Dec. 2002), 301–309. DOI: http://dx.doi.org/10.1109/TITS.2002.806806

[21] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar. 2001. Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18 /spl mu/. In *Proceedings of the Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No. 01CH37185)*. 61–62. DOI: http://dx.doi.org/10.1109/VLSIC.2001.934195

[22] D. S. Khudia and S. Mahlke. 2014. Harnessing soft computations for low-budget fault tolerance. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 319–330. DOI: http://dx.doi.org/10.1109/MICRO.2014.33

[23] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. 2013. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS'13)*. 31–40.

[24] M. Kooli and G. Di Natale. 2014. A survey on simulation-based fault injection tools for complex systems. In *Proceedings of the 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS'14)*. 1–6. DOI: http://dx.doi.org/10.1109/DTIS.2014.6850649

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. Retrieved from http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov. 1998), 2278–2324. DOI: http://dx.doi.org/10.1109/5.726791

[27] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR* abs/0810.1355 (2008). Retrieved from http://arxiv.org/abs/0810.1355.

[28] H. Li, D. Song, Y. Lu, and J. Liu. 2012. A two-view based multilayer feature graph for robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'12)*. 3580–3587. DOI: http://dx.doi.org/10.1109/ICRA.2012.6224732

[29] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO'09)*.

[30] T. Li, M. Shafique, J. A. Ambrose, S. Rehman, J. Henkel, and S. Parameswaran. 2013. RASTER: Runtime adaptive spatial/temporal error resiliency for embedded processors. In *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference (DAC'13)*. 1–7.

[31] J. W. Lichtman, H. Pfister, and N. Shavit. 2014. The big data challenges of connectomics. In *Nature Neuroscience*, volume 17. Nature Publishing Group, 1448–1454. DOI: http://dx.doi.org/10.1038/nn.3837

[32] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving DRAM refresh-power through critical data partitioning. *SIGPLAN Not.* 46, 3 (Mar. 2011), 213–224. DOI: http://dx.doi.org/10.1145/1961296.1950391

[33] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman. 2015. LLFI: An intermediate code-level fault injection tool for hardware faults. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*. 11–16. DOI: http://dx.doi.org/10.1109/QRS.2015.13

[34] A. Meixner, M. E. Bauer, and D. J. Sorin. 2008. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro* 28, 1 (Jan. 2008), 52–59. DOI: http://dx.doi.org/10.1109/MM.2008.3

[35] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. 2015. Doppelganger: A cache for approximate computing. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 50–61. DOI: http://dx.doi.org/10.1145/2830772.2830790

[36] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCS'10)*. 1–12. DOI: http://dx.doi.org/10.1109/HPCA.2010.5416635

[37] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. 2010. Quality of service profiling. In *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 25–34. DOI: http://dx.doi.org/10.1145/1806799.1806808

[38] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. 2002. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. 99–110. DOI: http://dx.doi.org/10.1109/ISCA.2002.1003566

[39] L. Murphy and P. Newman. 2011. Risky planning: Path planning over costmaps with a probabilistically bounded speed-accuracy tradeoff. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'11)*. 3727–3732. DOI: http://dx.doi.org/10.1109/ICRA.2011.5980124

[40] Michael A. Nielsen. 2015. *Neural Networks and Deep Learning*. Determination Press.
[41] N. Oh, S. Mitra, and E. J. McCluskey. 2002. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.* 51, 2 (Feb. 2002), 180–199. DOI: http://dx.doi.org/10.1109/12.980007
[42] H. Omar, M. Ahmad, and O. Khan. 2017. GraphTuner: An input dependence aware loop perforation scheme for efficient execution of approximated graph algorithms. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'17)*. 201–208. DOI: http://dx.doi.org/10.1109/ICCD.2017.38
[43] M. W. Rashid and M. C. Huang. 2008. Supporting highly-decoupled thread-level redundancy for parallel programs. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture*. 393–404. DOI: http://dx.doi.org/10.1109/HPCA.2008.4658655
[44] V. Reddy and E. Rotenberg. 2008. Coverage of a microarchitecture-level fault check regimen in a superscalar processor. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN'08)*. 1–10. DOI: http://dx.doi.org/10.1109/DSN.2008.4630065
[45] S. Rehman, F. Kriebel, Duo Sun, M. Shafique, and J. Henkel. 2014. dTune: Leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects. In *Proceedings of the 51st ACM/EDAC/IEEE Design Automation Conference (DAC'14)*. 1–6. DOI: http://dx.doi.org/10.1145/2593069.2593127
[46] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. 2005. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*. 243–254. DOI: http://dx.doi.org/10.1109/CGO.2005.34
[47] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. 2005. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.* 2, 4 (Dec. 2005), 366–396. DOI: http://dx.doi.org/10.1145/1113841.1113843
[48] Felipe Restrepo-Calle, Antonio Martínez-Álvarez, Sergio Cuenca-Asensi, and Antonio Jimeno-Morenilla. 2013. Selective SWIFT-R. *J. Electron. Test.* 29, 6 (Dec. 2013), 825–838. DOI: http://dx.doi.org/10.1007/s10836-013-5416-6
[49] V. Roberge, M. Tarbouchi, and G. Labonte. 2013. Comparison of parallel genetic algorithm and particle swarm optimization for real-time UAV path planning. *IEEE Trans. Industr. Informat.* 9, 1 (Feb. 2013), 132–141. DOI: http://dx.doi.org/10.1109/TII.2012.2198665
[50] E. Rotenberg. 1999. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*. 84–91. DOI: http://dx.doi.org/10.1109/FTCS.1999.781037
[51] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. ACM, New York, NY, 13–24. DOI: http://dx.doi.org/10.1145/2540708.2540711
[52] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. 2009. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. ACM, New York, NY, 122–132. DOI: http://dx.doi.org/10.1145/1669112.1669129
[53] I. Sato and H. Niihara. 2014. Beyond pedestrian detection: Deep neural networks level-up automotive safety. In *Proceedings of the GPU Technology Conference*.
[54] P. Sermanet and Y. LeCun. 2011. Traffic sign recognition with multi-scale convolutional networks. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'11)*. 2809–2813. DOI: http://dx.doi.org/10.1109/IJCNN.2011.6033589
[55] Muhammad Shafique, Semeen Rehman, Pau Vilimelis Aceituno, and Jörg Henkel. 2013. Exploiting program-level masking and error propagation for constrained reliability optimization. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*. ACM, New York, NY, Article 17, 9 pages. DOI: http://dx.doi.org/10.1145/2463209.2488755
[56] Q. Shi, H. Hoffmann, and O. Khan. 2015. A cross-layer multicore architecture to tradeoff program accuracy and resilience overheads. *IEEE Comput. Architect. Lett.* 14, 2 (July 2015), 85–89. DOI: http://dx.doi.org/10.1109/LCA.2014.2365204
[57] Q. Shi and O. Khan. 2013. Toward holistic soft-error-resilient shared-memory multicores. *Computer* 46, 10 (Oct. 2013), 56–64. DOI: http://dx.doi.org/10.1109/MC.2013.262
[58] Qingchuan Shi, Hamza Omar, and Omer Khan. 2017. Exploiting the tradeoff between program accuracy and soft-error resiliency overhead for machine learning workloads. *CoRR* abs/1707.02589 (2017). Retrieved from http://arxiv.org/abs/1707.02589.
[59] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, Lorenzo Alvisi, Ibm Technical, Contacts John Keaty, Rob Bell, and Ram Rajamony. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of International Conference on Dependable Systems and Networks*. 389–398.

[60] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, New York, NY, 124–134. DOI: http://dx.doi.org/10.1145/2025113.2025133

[61] T. J. Siegel, E. Pfeffer, and J. A. Magee. 2004. The IBM eServer Z990 microprocessor. *IBM J. Res. Dev.* 48, 3–4 (May 2004), 295–309. DOI: http://dx.doi.org/10.1147/rd.483.0295

[62] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *CoRR* abs/1409.1556 (2014). Retrieved from http://arxiv.org/abs/1409.1556.

[63] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. 2011. The german traffic sign recognition benchmark: A multi-class classification competition. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'11)*. 1453–1460. DOI: http://dx.doi.org/10.1109/IJCNN.2011.6033395

[64] Anthony Stentz. 1995. The focussed D* algorithm for real-time replanning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1652–1659.

[65] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. DSENT—A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of the International Symposium on Networks-on-Chip*.

[66] A. Vega, C. C. Lin, K. Swaminathan, A. Buyuktosunoglu, S. Pankanti, and P. Bose. 2015. Resilient, UAV-embedded real-time computing. In *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD'15)*. 736–739. DOI: http://dx.doi.org/10.1109/ICCD.2015.7357189

[67] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–14. DOI: http://dx.doi.org/10.1109/MICRO.2016.7783745

[68] R. Viguier, C. C. Lin, K. Swaminathan, A. Vega, A. Buyuktosunoglu, S. Pankanti, P. Bose, H. Akbarpour, F. Bunyak, K. Palaniappan, and G. Seetharaman. 2015. Resilient mobile cognition: Algorithms, innovations, and architectures. In *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD'15)*. 728–731. DOI: http://dx.doi.org/10.1109/ICCD.2015.7357187

[69] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. 2014. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14)*. 73–84. DOI: http://dx.doi.org/10.1109/ISCA.2014.6853227

[70] N. J. Wang and S. J. Patel. 2005. ReStore: Symptom based soft error detection in microprocessors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'05)*. 30–39. DOI: http://dx.doi.org/10.1109/DSN.2005.82

Received August 2017; revised January 2018; accepted April 2018