Evaluating the Error Resilience of Parallel Programs

Bo Fang*, Karthik Pattabiraman*, Matei Ripeanu*, Sudhanva Gurumurthi†
*Department of Electrical and Computer Engineering
University of British Columbia
Email: {bof, karthikp, matei}@ece.ubc.ca
†AMD Research, Advanced Micro Devices, Inc.
Email: sudhanva.gurumurthi@amd.com

***Abstract*—As a consequence of increasing hardware fault rates, HPC systems face significant challenges in terms of reliability. Evaluating the error resilience of HPC applications is an essential step for building efficient fault-tolerant mechanisms for these applications. In this paper, we propose a methodology to characterize the resilience of OpenMP programs using fault-injection experiments. We find that the error resilience of OpenMP applications depends on the program structure and thread model; hence, these need to be taken into account while characterizing error resilience. We also report preliminary results about the correlation between the application's error resilience and the algorithm(s) used in the application.**

I. Introduction

Prior studies [1]–[3] showed that fault rates experienced by processors and memory likely will increase as chips get denser. High-performance computing (HPC) systems usually have high reliability, availability and serviceability (RAS) requirements; thus, understanding the resilience characteristics of HPC applications is important for hardware and system/application designers to make sound decisions about how to design and provision them for desired RAS levels.

Our previous work [4] on studying the error resilience of GPGPU applications suggests that the resilience of an application correlates with its algorithmic characteristics. Among the twelve CUDA benchmarks we studied, we observed correlations between the algorithmic characteristic and the resilience of the applications. If our hypothesis that error resilience and algorithmic properties are correlated is true, then an analogous trend also should be observable when using programming models other than CUDA. This motivates us to continue to explore this space by investigating parallel applications that run on CPUs.

Several parallel-programming paradigms are used widely in HPC applications on CPUs. POSIX thread (pthread) and Open Multi-Processing (OpenMP) typically are used for shared-memory systems, while Message Passing Interface (MPI) typically is used for distributed-memory systems. Among these programming models, OpenMP is particularly interesting due to its strong emphasis on structured parallel programming [5]. Because OpenMP has gained a lot of attention in the HPC community, our study focuses on evaluating the error resilience of OpenMP parallel applications.

However, characterizing the error resilience of OpenMP applications is challenging due to two challenges:

1) Any error resilience characterization needs to consider the underlying thread model of the application. Because OpenMP programs contain two types of threads (master and slave threads) that accomplish different types of work, each thread's error resilience properties needs to be characterized separately in addition to its overall impact on application resilience;
2) Resilience characterization (e.g., through a fault-injection study) usually works at the assembly/machine-code instruction level, but we need source-code level information to understand the program structure and address the first challenge above.

To overcome these challenges, we perform static analysis of OpenMP programs using a modified version of the LLVM compiler [6]. We collect the necessary dynamic information from a program at runtime to obtain the execution profile of each of its OpenMP threads, thus addressing challenge 1. Based on this information, we map information from source-code level to the instruction level using LLVM, and conduct fault-injection experiments on specific program regions, thus addressing challenge 2.

This paper makes the following contributions:

1) Proposes a methodology to evaluate the resilience of OpenMP applications using fault-injection experiments,
2) Extends an existing fault-injection tool, LLFI [7] to support multi-threading programs (i.e. OpenMP),
3) Characterizes the error resilience of eight OpenMP parallel applications drawn from the Rodinia benchmark suite [8],
4) Explores the hypothesis that the error resilience of OpenMP programs correlate with their algorithmic characteristics

Our study's main findings are:

1) We find that the error resilience characterization of OpenMP programs needs to take into account the thread model and the program structure. Otherwise, the resilience characteristics could be biased (as shown in Figure 4 and Figure 6). This is important as different threads and structures of OpenMP programs show different level of resilience, which opens the opportunity to use differentiated fault-detection and impact-mitigation mechanisms to reduce the error detection overhead and improve resilience. Our characterization mechanisms are

insightful for studying the fault tolerance of parallel programs, and instructive to design future hardware fault detection techniques.

2) We find preliminary support for our hypothesis that error resilience properties do correlate with the algorithmic characteristics of parallel applications. If this hypothesis is indeed true, then this opens an avenue to understand application resilience at much lower cost without conducting exhaustive and complex fault-injection experiments. This can also provide useful information on designing algorithmic specific error detection and recovery mechanisms.

II. Related Work

This section provides an overview of related work in the areas of error resilience studies of parallel programs and explains how our work differs.

Fault injection has been well-explored on CPUs using run-time debuggers. Examples are GOOFI [9] and NFTAPE [10]. However, they do not consider multi-threaded programs, nor do they concern themselves with choosing different parts of a program for injection. Other work [11] attempted to inject faults in scientific applications using the PIN tool from Intel, a dynamic binary instrumentation framework, but multi-threaded fault injection was not their focus at that time.

Lu et al. [12] proposed a way to assess fault sensitivity in MPI applications by injecting faults into registers, application memory regions and messages at run-time. They showed that registers and MPI messages are particularly vulnerable to single-bit-flip faults and urge the MPI standard to redesign and enhance the error-handling APIs in the context of hardware faults. While their goals were similar to our study, their study did not consider shared-memory parallel programs, which are very different from MPI programs.

Wei et al. [13] focused on leveraging similarities between threads in parallel programs to protect faults in program's control data. They used LLVM compiler infrastructure to instrument a parallel program with fault-detecting code and the PIN tool to conduct fault injections to evaluate its detectors. Their methodology integrated software level instrumentation and assembly-level evaluation. However, their study was based on pthread programs, which differ from OpenMP in terms of programming models and code structures. In addition, their evaluation was about only the control data and did not study end-to-end program vulnerability.

Sloan et al. proposed algorithmic approaches [14] to locate errors during the execution and partially recompute the result on parallel systems. They improved the performance of the Conjugate Gradient solver in high-error scenarios by 3x-4x and increased the probability that it completes successfully by up to 60%. Their results showed that the application-specific techniques help improve the fault tolerance of applications. However, their techniques were based on mathematical methods, which can be applied only to linear algebra problems such as matrix-vector multiplication. Our goal in contrast is to find techniques for general-purpose parallel applications.

To the best of our knowledge, it is the first to study experimentally the error resilience of OpenMP programs, and discuss the possibility of correlating the resilience of applications with their algorithm characteristics.

III. Methodology

This section introduces our methodology. As we have discussed in Section I, two factors may have unequal impacts on the resilience of OpenMP programs; faults in different types of threads and faults in different parts of the program. Therefore, it becomes important to study these factors separately.

Figure 1 shows the thread model of OpenMP programs. An OpenMP application starts as a single thread, which is the master thread. As the program executes, it may encounter one or more parallel regions, at which point slave threads are created by the master thread and run in parallel (including the master thread). Therefore, the master thread and slave threads differ from each other in terms of their behaviors and amounts of work performed. Our methodology takes into account the thread model of OpenMP programs. In addition to the thread model, we need to identify the program structure of the master thread. As shown in Figure 2, we partition the entire execution of the master thread into five segments. Each segment represents a task that the master thread does within that part of the code. For example, pre-algorithm and post-algorithm are segments that respectively contain the pre- and post- processing of the input and output data for the parallel region.

To make the preceding description more concrete, Figure 3 shows a code-snippet from the srad application, a diffusion method for ultrasonic and radar- imaging applications based on partial differential equations. The code snippet presents the main stages of srad, namely (1) image reading, (2) pre-processing (resizing, setting up and scaling down), (3) computation, (4) scaling up, and (5) output. The figure also shows how the stages correspond to the segments identified in the preceding paragraph. We believe that understanding the impact of each segment on the resilience of the program is important for understanding its overall error resilience.

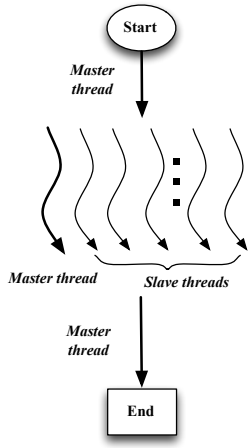

Fig. 1: An example of the thread model of OpenMP programs

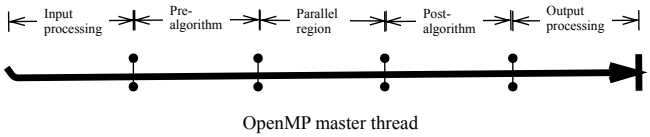

Fig. 2: Program segments of the master thread

```
Begin:                                                          Input
    //read image                                                processing
    read_graphics(image_orig);
    //resize the image
    resize(image_orig);
    //setup
    image = image_setup(image_orig);                            Pre-
    //image scale down                                          algorithm
    for (i=0; i<Ne; i++){
      image[i] = exp(image[i]/255);
    }
    //computation
    #pragma omp parallel for shared(var_list1)
private(var_list2)
    for (j=0; j<Nc; j++){
    //directional derivatives,ICOV,diffusion coefficient        Parallel
    }                                                           region
    #pragma omp parallel for shared(var_list1)
private(var_list2)
    for (j=0; j<Nc; j++){
      //divergence & image update
    }
    //image scale up                                            Post-
    for (i=0; i<Ne; i++){                                       algorithm
      image[i] = log(image[i])*255;
    }
    //write image out                                           Output
    write_graphics(image);                                      processing
    free_memory()
End
```

Fig. 3: An example of the partition of an OpenMP program

A. Fault model

Hardware faults can be classified broadly as transient or permanent. Transient faults usually are "one-off" events and occur non-deterministically, while permanent faults persist at a given location. Further, transient faults are caused by external events such as cosmic rays and over-heated components, while permanent hardware faults usually are caused by manufacturing or design faults. We consider only transient faults in this study. We use the single-bit-flip model in this study because it is the *d*e-facto fault model adopted in studies of transient faults [13], [15], [16]. However, our fault injector can support both single- and multiple-bit flips. We will consider multiple-bit errors in future work.

We consider faults in the CPU's computational elements. We inject faults into the destination register of the program's instructions to simulate an error in the processor's computational units. For memory instructions, we inject faults into address-calculation parts in load/store instructions. We assume that the memory is error-correction-code (ECC)-protected, as is typically the case in HPC systems, and so do not inject faults into memory values.

An application may have four outcomes after a fault is injected:

1) Throws an exception (*crash*)
2) Times out by going into an infinite loop (*hang*)
3) Completes with incorrect output (*silent data dorruption (SDC)*) [1]
4) Completes with correct output (*benign*).

These four outcomes are mutually exclusive and exhaustive. Among the four outcomes, SDC is the most serious one because there is no indication that the result is incorrect. Therefore, we focus on SDC in our evaluation of a program's error resilience in this study.

B. Fault-injection tool

We extend the fault-injection tool called LLFI to inject faults into OpenMP applications [7]. LLFI performs fault injection at the LLVM compiler's intermediate code level. It instruments the original intermediate representation (IR) of an application with fault-injection code, which performs the actual injection at run-time. LLFI allows users to specify the kinds of faults and locations (i.e., instructions, operands) to inject. Although LLFI is a convenient tool for fault injection, it did not support multi-threaded programs. We extended LLFI and redesign the interface between its components to support multi-threading.

The operation of LLFI consists of three main phases, all of which we modify for multi-threaded programs as follows:

1) *Instrumentation phase* adds callback functions after each IR instruction. The instructions to be instrumented are determined based on the configurations specified by users. We add new configurations in this phase to allow the user to specify individual regions and threads as targets for instrumentation.
2) *Profiling phase* uses the callback functions added in the first phase for counting and generating statistics, such as the total number of dynamic instructions of the program. We modify these functions to keep the statistics on a per-thread basis. The statistics are used in the next phase to choose a random instruction into which to inject .
3) *Injection phase* uses the callback functions added in the first phase, and the statistics added in the second phase to inject faults. To inject a fault, a random instruction is chosen from the set of all dynamically executed instructions in the program and a single bit is flipped in its address operand (for memory instructions) or destination register (otherwise). Our modification to this phase consists of performing the injections on a per-thread basis using the per-thread statistics in the second phase, and into specific program regions as described in the first phase.

IV. Error Resilience Characteristics

We run our experiments on an Intel Xeon CPU X5650 multi-core processor running at 2.67GHz with 24 hardware cores. The standard distribution of Clang, which is a compiler frontend in the LLVM tool-chain, does not support OpenMP directives, so we use an implementation of the OpenMP

[1] We define an SDC as an outcome that differs from the fault-free run.

C/C++ language extensions in Clang from [17]. We use eight OpenMP programs from the Rodinia benchmark suite [8] in our evaluation. We configure the OpenMP programs with 24 threads, which is equal to the number of cores. For each benchmark, we conduct more than 10,000 fault-injection runs. In each run, we inject a single random fault using our extended version of LLFI.

A. Thread-level Differences on Error Resilience

We evaluate the effects of different kinds of threads on the error resilience of the program. We perform two separate experiments for injecting faults into the master and slave threads respectively. In the first experiment, the master thread (threadID = 0) is chosen and the locations to inject are distributed over the entire execution of the program. In the second experiment, a random slave thread is chosen and the locations to inject are confined to the parallel segment of the OpenMP program.

This set-up is based on the following hypothesis: threads within the parallel segment resemble each other in OpenMP programs, and so choosing a random thread into which to inject is sufficient. We test our hypothesis by counting the number of dynamic instructions in each thread as a way to represent the behaviors of threads. Other possible solutions to determine the similarity of threads are still under-explored. We report the mean value and the standard deviation of the number of instructions in all threads of each benchmark in Table I. Our results indicate that most of the benchmarks show similar behaviors across threads within the parallel segment, as demonstrated by the relatively low standard deviation. The one exception is the nn (k-nearest neighbours) benchmark, which exhibits high standard deviation (SD). This is because the 24 threads of nn are clustered into two groups (one with 10 threads and the other with 14) according to the number of instructions they execute. The probability of choosing a thread from the second group is higher by roughly 8% (1/12). We disregard this difference for simplicity.

Figure 4 presents the SDC rates for two sets of experiments. In most of the benchmarks, the master thread has a higher SDC rate than slave threads except for lud. This is lud's master thread spends most of its time in the parallel segment, and hence it makes no difference whether we inject into the master thread or into slave threads. The average deviation of the SDC rate between threads is 7.8%, with the biggest deviation of 16% in pathfinder. This suggests that performing a naive fault injection without taking thread-level differences into account can grossly misestimate the error resilience of applications.

To add to this point, we profile an OpenMP version of matrix multiplication (mm), which is not included in Rodinia benchmark suite. Of a total of 24 threads, only five of them (on average, executing 89476 dynamic instructions) perform significant amount of work, while the other 19 finish rather quickly (after executing 22 instructions). A random thread fault injection on mm without considering thread deviation leads to about 5% SDC rate, compared to performing fault injection on the master thread, which has an SDC rate of 28%.

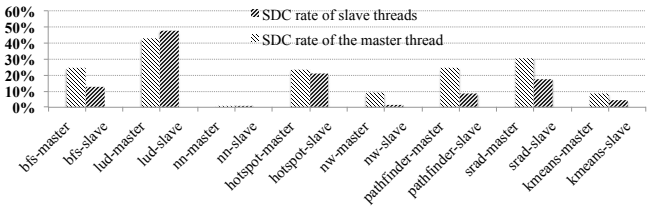

Fig. 4: Differences in the SDC rates of applications between injections in the master thread and slave threads

B. Segment-level Differences on Error Resilience

Our solution for evaluating the resilience of OpenMP programs more accurately is to consider the resilience of the master thread and slave thread separately. For slave threads, we need to consider only the parallel segment in which slave threads are spawned. However, the master thread consists of multiple segments, and hence we need to consider the error resilience of each segment separately. To do that, we first manually split the each benchmark into segments and measure the time spent in each segment. We intend to automate this process in the future. Figure 5 shows the execution time of each segment of the benchmarks. As we can see, the parallel segment dominates the execution time in six of the programs, while I/O operations (output processing in these cases) take longer time than the other segments in two programs (bfs and srad).

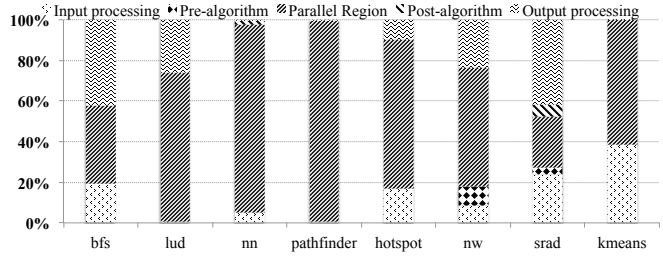

Fig. 5: Execution time profile of the OpenMP benchmarks

Ideally, fault injection should be based on the execution time of each segment as measured by a consistent metric such as in cycles. Because we are limited by the LLFI infrastructure, which lacks information about the underlying execution of the program, our fault injection is based on the number of dynamic instructions executed in each segment. Our injection proceeds as follows. First, we manually map each segment in the source code to IR code, and find the corresponding instructions that represent the boundaries in the IR code. We use this information to find the IR instruction range for each segment. Knowing these boundaries, when we inject a fault using LLFI at the IR code level, we can tell when the fault is injected and which segment the program is executing at the time of the injection.

Figure 6 shows the SDC rate of each segment in seven of the eight benchmarks. We did not show nn because the overall SDC rate of nn is quite small compared to other benchmarks (less than 1%). We also skip the output segments

TABLE I: Mean and the standard deviation of the number of dynamic instructions of threads

Benchmark	bfs	lud	nn	pathfinder	hotspot	nw	srad	kmeans
Mean	42,505	3,568,024	327,652	31,320	14,275	6,540,148	1,993,166	4,150,695
Standard deviation	796	292,682	178,752	364	2,568	44,123	29,487	383,953

TABLE II: Mean and standard deviation of SDC rate in each segment of the programs

Segment	Input processing	Pre-algorithm	Parallel segment	Post-algorithm	Output processing
Number of applications	6	2	7	2	5
Mean	28.59%	20.32%	16.13%	27%	42.42%
Standard deviation	11%	23%	14%	13.2%	5.1%

[a] The first row shows the count of number of applications that contain the segment

of the nw and pathfinder applications, because the time spent in those segment is less than 1% of the total time, which means that the chance that faults happen during the execution of those segments is small. We quantify the mean and standard deviation for each segment as shown in Table II.

We find that output processing segments exhibit much higher average SDC rates than other segments. This is intuitive because output processing is close to the end of the execution, and so faults are more likely to affect the final output. We also find that the standard deviations of SDC of the pre-algorithm, post-algorithm and parallel segments are higher than the corresponding standard deviations of SDCs in the input processing and output processing segments. This shows that the resilience of the input and output processing segments is more stable than the algorithm-related segments across the applications.

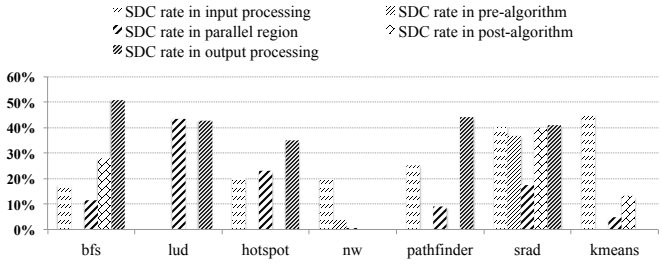

Fig. 6: Differences in the SDC rates in different segments of the benchmarks

Finally, to estimate the overall resilience, we propose two approaches depending on the goal of the resilience characterization. In the first approach, we consider the SDC rate of each segment separately and combine it with its time profile, to obtain the overall SDC rate. In the second approach, we consider only the algorithm-related segments, (i.e., pre- and post- algorithm and parallel segments). The results for both cases are shown in Figure 7. For the first case, the average SDC rate is 20%; for the second case, it is 14%. In both cases, the highest SDC rate is from lud and the lowest is from nn.

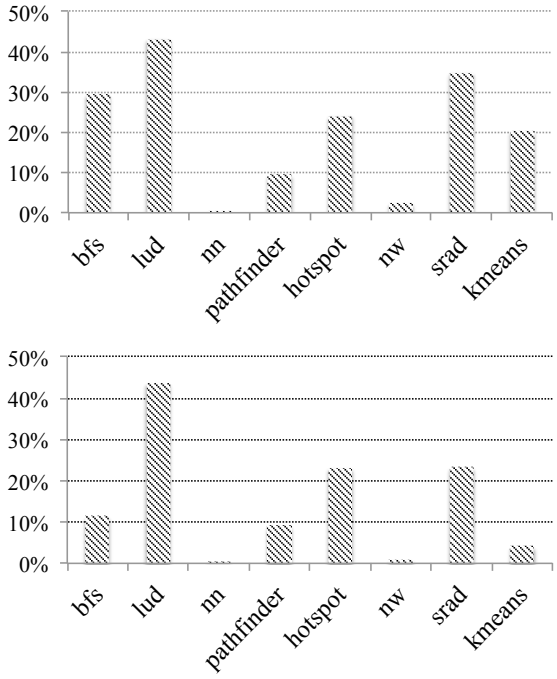

Fig. 7: SDC rates estimated in two approaches. *Top*: End-to-end SDC rates considering all segments *Bottom*: SDC rates when considering only algorithm-related segments excluding the input and output processing segments

V. Discussion

Our previous study on the error resilience of GPGPU applications [4] suggests that the algorithmic characteristics of the application influence its error resilience. The algorithmic operations we consider are described in Table III. For example, comparison-based operations, such as those used for searching for keys or in min/max value computation, have much higher error resilience than other operators. As another example, ”average-out” operations in which the final result is based on iterative computation over the previous results similarly have high resilience.

TABLE III: Operations that may affect resilience of GPGPU applications

Operations	Description
Comparison-based	Comparing two values, High resilience as the likelihood of maintaining the correct value is high
Average-out	The final state is a product of multiple temporary states, usually including iteration and merging
Graph processing	Graph-related algorithms such as breadth-first search
Linear algebra	Combination of basic linear algebra computation
Bit-wise operation	Input data are chunked based on certain length in bits

One of the goals of this work is to test the hypothesis that the error resilience of an application is correlated with

its algorithmic operations. If the hypothesis is true, then we should observe a trend in OpenMP programs similar to the tread we observed in GPGPU programs. To this end, we analyze the algorithms of eight benchmarks to identify the algorithmic operations performed. Table IV lists the high-level classification (dwarf) of these benchmarks from the Rodinia homepage, and the operations that we found in each benchmark.

We identify a total of five operations that may affect the error resilience of the application, four of which are also found in GPGPU applications: comparison-based, graph processing, linear algebra and average out. The fifth operation we introduce is grid-structured computation, which consists of regular or irregular grid calculations such as stencil computations.

TABLE IV: Operations of OpenMP benchmarks and the estimation of the resilience

Operations	Benchmarks	Observed SDC	Application dwarfs
Comparison-based	nn, nw	$0.07\% \sim 1\%$	Dynamic programming, dense linear algebra
Grid computation	hotspot, srad	23%	Structured grid
Graph processing	bfs,pathfinder	$9\% \sim 10\%$	Graph traversal, dynamic programming
Average-out	kmeans	4.2%	Dense linear algebra
Linear algebra	lud	44%	Dense linear algebra

Table IV shows the observed SDC rates for the OpenMP applications grouped by the algorithmic operations. Among the five operations we considered, comparison-based and average-out operations are the most resilient and linear algebra operations are the least resilient. This matches what we found in GPGPU applications for the four shared categories, suggesting that algorithmic operations may be an important factor in understanding the error resilience of parallel programs on both CPUs and GPUs. Based on our preliminary study, we intend to explore this space more systematically in the future.

VI. Summary

This paper presents a methodology to investigate the end-to-end error resilience characteristics of OpenMP applications through fault injection. This methodology overcomes the challenges in building a fault injector for OpenMP applications by taking into account the thread model and program structure of OpenMP applications. We extend an existing fault-injection tool, LLFI, to support multi-threading and inject faults into eight OpenMP programs from the Rodinia benchmark suite.

Our experiments show that, on average, 14% of the injected faults result in SDC when only the algorithm-related parts of the code are considered, while 20% of the injected faults result in SDC when the entire program (including input and output processing) is considered. We also find significant variations in SDC rates depending on the thread and program segment into which the fault is injected. Finally, we find preliminary evidence that the algorithmic characteristics of an application are correlated with its observed SDC rates. While these results are preliminary, they corroborate with our earlier results for GPGPU applications, which showed that these correlations might not be platform-specific.

Acknowledgment

This work was supported in part by an NSERC Discovery grant, an NSERC Engage Grant and a research gift from AMD Corporation. We thank the anonymous reviewers of FTXS2014 for their feedback to improve the paper.

References

[1] S. Michalak, A. Dubois, C. Storlie, H. Quinn, W. Rust, D. DuBois, D. Modl, A. Manuzzato, and S. Blanchard, "Assessment of the impact of cosmic-ray-induced neutrons on hardware in the roadrunner supercomputer," *Device and Materials Reliability, IEEE Transactions on*, vol. 12, no. 2, pp. 445–454, June 2012.

[2] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: A large-scale field study," in *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '09, 2009.

[3] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012.

[4] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014.

[5] G. J. Barbara Chapman and R. van der Pas, *Using OpenMP*. 55 Hayward Street, Cambridge, Mass, USA: The MIT press, 2007.

[6] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar 2004.

[7] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, June 2014.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[9] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Goofi: generic object-oriented fault injection tool," in *Dependable Systems and Networks, 2001 International Conference on*, 2001, pp. 83–88.

[10] D. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R. Iyer, "Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *IPDPS 2000*, 2000, pp. 91 –100.

[11] D. Li, J. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, 2012, pp. 1–11.

[12] C. da Lu and D. Reed, "Assessing fault sensitivity in mpi applications," in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, Nov 2004, pp. 37–37.

[13] J. Wei and K. Pattabiraman, "BLOCKWATCH: Leveraging similarity in parallel programs for error detection," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2012.

[14] J. Sloan, R. Kumar, and G. Bronevetsky, "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.

[15] K. S. Yim, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *IEEE International Parallel and Distributed Processing Symposium*, 2011.

[16] W. Gu, Z. Kalbarczyk, and R. Iyer, "Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors," in *Dependable Systems and Networks, 2004 International Conference on*, 2004, pp. 887–896.

[17] [Online]. Available: http://clang-omp.github.io/