Investigating Resilience of Loops in HPC Programs: A Semantic Approach with LLMs

Hailong Jiang[1*], Jianfeng Zhu[1], Bo Fang[2], Chao Chen[3] and Qiang Guan[1*]
[1]*Department of Computer Science, Kent State University, Kent, OH, USA*
{hjiang13, jzhu10, qguan}@kent.edu
[2]*Pacific Northwest National Laboratory, Richland, WA, USA*
bo.fang@pnnl.gov
[3]*Intel Corporation, Santa Clara, CA, USA*
chao.chen@intel.com

2024 IEEE High Performance Extreme Computing Conference (HPEC) | 979-8-3503-8713-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPEC62836.2024.10938472

***Abstract*—Transient hardware faults, resulting from particle strikes, are significant concerns in High-Performance Computing (HPC) systems. As these systems scale, the likelihood of soft errors rises. Traditional methods like Error-Correcting Codes (ECCs) and checkpointing address many of these errors, but some evade detection, leading to silent data corruptions (SDCs). This paper evaluates the resilience of HPC program loops, which are crucial for performance and error handling, by analyzing their computational patterns, known as the thirteen dwarfs of parallelism. We employ fault injection techniques to quantify SDC rates and utilize Large Language Models (LLMs) with prompt engineering to identify the loop semantics of the dwarfs in real source code. Our contributions include defining and summarizing loop patterns for each dwarf, quantifying their resilience, and leveraging LLMs for precise identification of these patterns. These insights enhance the understanding of loop resilience, aiding in the development of more resilient HPC applications.**

***Index Terms*—Resilience, Soft errors, Loops, HPC programs, Large Language Models**

I. Introduction

Transient hardware faults, often resulting from particle strikes, are significant concerns in High-Performance Computing (HPC) systems [1]. As HPC systems continue to scale, the likelihood of soft errors also rises [2]. While hardware-level and system-level mechanisms, such as Error-Correcting Codes (ECCs) and checkpointing/restart schemes, can detect and correct many soft errors, some errors evade these measures and propagate to applications [3]. These undetected errors can lead to application failures and severe outcomes, including silent data corruptions (SDCs). Therefore, implementing resilience approaches to ensure the correctness of HPC programs is crucial.

The duplication technique is a space-time trade-off method that executes the program twice to compare results and detect errors [4]. However, the duplication approach requires up to twice the resources. The challenge of applying duplication in HPC programs is to reduce the volume of duplication needed. Through experiments, Hussain [5] demonstrates that partial duplication for HPC programs typically yields higher performance under different node failure rates compared to full duplication. However, the partial duplication scheme necessitates an understanding of which parts of the program are more error-prone.

HPC applications typically consist of many loops. A main computation loop in a scientific simulation workload often dominates the execution time. Within this main loop, several inner loops are typically used to update large data objects (e.g., a mesh structure in computational fluid dynamics) and perform iterative computations to determine the properties of these objects, such as the energy of particles. Since loops in HPC programs consume the most execution time and resources, it is reasonable to allocate limited duplication resources to the most critical loops. However, there remains a gap in implementing partial duplication for loops: determining the vulnerability ranking of loops within a program.

Previous research has demonstrated that the semantics of code significantly influence the resilience of HPC programs by affecting error propagation [6]. B. Fang et al. [7] showed that the variability in the SDC rate is closely related to the computational patterns of applications. Their study categorized benchmarks into five resilience categories based on the 'seven dwarfs of parallelism,' which are common computational patterns in parallel applications [8]. They found that benchmarks within the same dwarf exhibit similar SDC rates. By measuring the SDC rates, Fang et al. [7] quantified the resilience of specific computational patterns. Since these dwarfs are primarily realized through loops, identifying loops that correspond to specific dwarfs enables the prediction of the SDC rate for those loops. However, they did not establish a mapping between loop patterns and

To improve scalability across diverse codebases and leverage advanced natural language processing techniques for more precise and automated analysis of these patterns, we introduce Large Language Models (LLMs), especially those based on the transformer architecture like GPT-4 [9]. Recent studies [10]–[15] have demonstrated the effectiveness of LLMs in various programming tasks (e.g., code generation,

Thanks to the support partially from NSF #2217104 #2212465.

translation, completion), yielding impressive results. Pioneering studies [13], [16] have begun to explore the application of LLMs in the HPC domain, such as parallel code generation, indicating a promising synergy between large language models and HPC [17]. A recent survey [18] highlights the need for more challenging tasks to evaluate LLMs' reasoning abilities, particularly in vulnerability detection. This capability has profound implications for potential applications of LLMs in other tasks requiring understanding and reasoning about code semantics. Furthermore, LLMs are ushering in a new era of prompt engineering, which demands new skills from software engineers. Researchers have proposed methods like Automatic Semantic Augmentation of Prompts for software engineering tasks [19].

While prior work has established a link between code semantics and resilience, no studies have specifically assessed loop resilience in HPC programs through a semantic approach using LLMs. This paper aims to fill the gap by evaluating the resilience of loops from a semantic perspective. We aim to answer the following research questions:

- RQ1: How can computational patterns, known as dwarfs, be identified from source code using their loop semantics?
- RQ2: How do SDC rates vary among HPC computational patterns, and what does this reveal about their resilience to soft errors?
- RQ3: How can Large Language Models (LLMs) be used to accurately identify the loop semantics of the 13 dwarfs in real HPC applications?

To answer these questions, we need to undertake several key steps. First, we must define and summarize the loop patterns that correspond to each of the thirteen dwarfs of parallelism. Second, we will employ fault injection (FI) techniques to quantify the SDC rates of each of the thirteen dwarfs. Lastly, we will leverage Large Language Models (LLMs) with prompt engineering to identify these dwarfs in real source code. This approach will enable us to automatically detect and categorize loop patterns within HPC applications, facilitating the prediction and enhancement of their resilience to soft errors.

Our work makes the following contributions:

- **Mapping between Loop Patterns and Dwarfs**: We systematically define and summarize the loop patterns associated with each of the thirteen dwarfs of parallelism.
- **SDC Rate Inspection across Computational Patterns**: We measured the SDC rates of thirteen computational patterns (dwarfs) to provide a detailed analysis of their resilience to soft errors.
- **Identification of Dwarfs by LLMs**: Our language models (LLMs), with prompt engineering, accurately identify and categorize loop patterns within real source code and IR code compiled from LLVM.

The rest of this paper is organized as follows: Section II covers background information. Section III summarizes the patterns and loops associated with the thirteen dwarfs of parallelism. Section IV details our methodology for SDC rate analysis across computational patterns. Section V describes our approach to identifying dwarfs in real source code using Large Language Models (LLMs) with prompt engineering. Section VI discusses our findings and implications, and Section VII presents our conclusions.

II. Background

In this section, we describe the background of loops, the terms related to resilience, the fault model, and the concept of loop semantics.

A. Fault Model

In this paper, we consider soft errors that occur in the computational elements of the processor, including the pipeline register and functional units. We do not consider faults in the memory or caches, as we assume these are protected with error correction codes (ECC). The study by Sangchoolie et al. [20] demonstrates that the consequences of multi-bit errors are similar to those of single-bit errors, leading us to prioritize single-bit errors in our analysis. Additionally, we focus on single-bit errors to ensure clarity and precision in our research approach. Our fault model is in line with other work in the area [21]–[23].

B. Common Metrics for Measuring Resilience

Fault Injection (FI) [24]–[32] is the dominant technique for evaluating the resilience of applications against errors in HPC systems. This method involves numerous random fault injections, each of which randomly targets an instruction and triggers bit flips in the input or output operands during the application's execution. The results of these injections are classified into three categories:

- **Benign:** The program output matches the error-free execution;
- **Silent Data Corruption (SDC):** The output deviates from the error-free result, but the program does not terminate;
- **Crash:** The error causes the operating system to terminate the program.

The application's resilience is then measured primarily by assessing the SDC rate with

$$R_{SDC} = \frac{\#\text{ of SDC instances}}{\#\text{ of FIs}} \tag{1}$$

In this study, we primarily focus on Silent Data Corruption (SDC) rates, as SDCs represent the most concerning form of error in HPC applications. SDCs are particularly insidious because they cause incorrect data to be produced without detection, leading to potentially significant inaccuracies in computational results.

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

C. Loop Terminology

A loop is a sequence of instructions that is continually repeated until a certain condition is met. In programming, loops are used to perform repetitive tasks efficiently.

There are three types of loops in HPC programs:

1) For Loops: Iterate a set number of times, often used for processing arrays or matrices.

2) While Loops: Continue to iterate as long as a specified condition remains true.

3) Do-While Loops: Execute the loop body at least once before testing the condition.

D. Concept of Loop Semantics

1) Computational Patterns: Computational patterns are recurring structures in algorithms that solve common problems in parallel computing. Understanding these patterns helps optimize code for performance and resilience in HPC applications.

2) Dwarfs: The 13 Dwarfs of parallel computing were introduced by the Berkeley View team [8]. Each dwarf represents a high-level computational pattern that captures the essence of a broad class of applications, as listed in Table I.

3) Loop Semantics: Loop semantics refers to the specific computational behavior and purpose of loops within a program, often aligning with recognized computational patterns. Loops are the primary constructs that implement the computational patterns represented by the dwarfs. By analyzing loop semantics, we can categorize loops into these dwarfs. Different loop semantics (or computational patterns) exhibit varying resilience characteristics.

4) Relationships Among Computational Patterns, Dwarfs, and Loop Semantics: Computational patterns are high-level algorithmic structures that solve common problems in parallel computing, represented by the 13 dwarfs. Loop semantics describes the specific behaviors and purposes of loops that implement these computational patterns, directly influencing the resilience of HPC applications.

III. Semantic Analysis of Loops

In this section, we focus on understanding the resilience of loops based on their semantic characteristics, particularly by examining their association with common computational patterns known as dwarfs. By categorizing loops according to these dwarfs, we aim to establish a connection between the semantic properties of loops and their resilience. This analysis, although not exhaustive, provides valuable insights into how different types of computational patterns affect the robustness of loops in HPC programs.

A. Identification of Loop Semantics

To identify the semantic characteristics of loops, we categorize them based on their alignment with the well-established computational patterns known as dwarfs. These dwarfs represent common motifs in high-performance computing, as listed below.

Each of these patterns embodies specific computational behaviors that influence a loop's resilience to soft errors. By classifying loops according to these patterns, we can systematically analyze their semantic properties and draw connections to their resilience characteristics.

We propose the following ideas to identify the dwarfs by loop semantics:

1) **Dense Linear Algebra**: Operations on dense matrices and vectors.
   - Look for loops performing matrix and vector operations, such as matrix-matrix multiplication, matrix-vector multiplication, or linear transformations.
2) **Sparse Linear Algebra**: Operations on sparse matrices and vectors.
   - Identify loops working on sparse matrices and vectors, typically involving conditionals to handle non-zero elements.
3) **Spectral Methods**: Computation of Fourier transforms and related operations.
   - Search for loops implementing Fourier transforms or other spectral techniques.
4) **N-Body Methods**: Simulations involving a large number of interacting particles.
   - Detect loops simulating interactions between a large number of particles, such as gravitational or molecular dynamics simulations.
5) **Structured Grids**: Computations on regular, structured grids.
   - Identify loops iterating over regular, structured grids, commonly used in finite difference or finite volume methods.
6) **Unstructured Grids**: Computations on irregular, unstructured grids.
   - Recognize loops operating on irregular, unstructured grids, typical in finite element methods.
7) **Map Reduce**: Processing large data sets with a distributed algorithm on a cluster.
   - Look for map and reduce operations, often implemented as separate loops for processing data chunks and aggregating results.
8) **Combinational Logic**: Operations on combinational circuits.
   - Identify loops performing bitwise operations, logic gates, or simple arithmetic operations.
9) **Graph Traversal**: Operations on graphs, including search algorithms.
   - Search for loops iterating over graph nodes and edges, implementing algorithms like breadth-first search, depth-first search, or shortest path.
10) **Dynamic Programming**: Solving problems by

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

breaking them down into simpler subproblems.
    - Identify nested loops filling in a table of solutions to subproblems, characteristic of dynamic programming algorithms.
11) **Backtrack and Branch-and-Bound**: Techniques for combinatorial optimization problems.
    - Recognize loops generating and testing solution candidates, often involving recursive calls and backtracking.
12) **Finite State Machines**: State transition-based computations.
    - Look for loops iterating over input events or characters, managing state transitions within the loop body.
13) **Graphical Models**: Probabilistic models representing complex dependencies among variables.
    - Identify loops updating probabilities or performing inference in probabilistic models like Bayesian networks or Markov random fields.

More detailed information is listed in Appendix A. We mapped each core loop in the selected HPC benchmarks to one of these categories based on its computational pattern.

IV. Analysis of SDC Rates Across Computational Patterns

To quantify the resilience of the computational patterns represented by the 13 dwarfs, we conducted a series of fault injection experiments and statistical analyses. This section outlines the methodology used to measure and analyze the resilience of these patterns.

A. Statistical Analysis

Benchmark Selection: We selected a diverse set of HPC benchmark programs, ensuring that each of the 13 dwarfs was well-represented. The benchmarks included EEMBC, SPEC2006, Rodinia [33], NPB [34], and LULESH [35].

Fault Injection Setup: Our fault injection (FI) tool of choice is LLFI [36], a low-level fault injector that operates at the LLVM (LLVM 13) Intermediate Representation (IR) level. LLFI leverages the LLVM compiler to translate programs written in high-level programming languages like C/C++ into LLVM IR. For each benchmark application, we conducted 3,000 fault injection runs to generate a comprehensive dataset.

The FI experiments were conducted on a Dell Workstation equipped with 32 Intel(R) Xeon(R) CPUs E5-2620 v4 @ 2.10GHz. The hardware operates on x86-64 architectures with a 64-bit system. We adapted LLFI for LLVM 13, running on Ubuntu 18.04.

Measurement of Resilience: Resilience was quantified by measuring the rates of Silent Data Corruptions (SDCs). The SDC rates represent the resilience at the application level. Since the dwarfs occupy the majority of the execution time, the application-level resilience can be considered equivalent to the dwarf-level resilience.

B. Results and Interpretation

Table I presents a detailed summary of Silent Data Corruption (SDC) rates across various computational patterns, referred to as "dwarfs," and their benchmarks in High-Performance Computing (HPC). The SDC rates represent the resilience of these computational patterns to soft errors, providing valuable insights into their susceptibility to such faults.

We refer to the results of **Sparse** and **Dense Linear Algebra**, **Structured Grids** and **Unstructured Grids**, **Map Reduce**, **Backtrack and Branch-and-Bound**, and **Graphical Models** from [7]. The detailed analysis has been described in [7]. The results from our experiments are comparable to theirs since we used the same LLFI tool for fault injection and adhered to the same definition of SDC.

Combinational Logic exhibits the highest SDC rate. This high susceptibility is observed in applications involving hashing (SPEC 2006 CRC) and RSA, where the intensive use of bitwise operations and logical gates increases the likelihood of undetected errors.

N-Body Methods, with SDC rates between 30% and 35%, are highly susceptible to soft errors. Exemplified by CoMD [41] and Lulesh [35], this high rate can be attributed to the complex interactions and dependencies between particles in these simulations, where errors in one part of the computation can easily propagate and magnify throughout the system.

Finite State Machines exhibit an SDC rate of 23%, as seen in SPEC2006 Integer: Text processing (perlbench). They are moderately resilient due to structured state transitions that help contain errors.

Dynamic Programming exhibits a relatively high SDC rate of 30%. Errors in previously computed subproblem solutions can easily propagate, affecting overall resilience.

These results show that variability in the SDC rate is related to the applications' characteristics. For example, N-Body Methods and Dynamic Programming, with their high SDC rates, require more robust error detection and correction mechanisms. Conversely, MapReduce and Backtrack and Branch-and-Bound, with their lower SDC rates, can benefit from less intensive resilience techniques. Future research should focus on refining these strategies, considering the specific resilience profiles of each dwarf. Additionally, exploring the use of advanced AI techniques, such as Large Language Models (LLMs), to predict and enhance loop resilience could provide significant advancements in the field.

V. Semantic Identification by LLMs

Instruction duplication is a method to protect application execution. Due to its high cost, selecting

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

TABLE I
SDC RATES ACROSS DIFFERENT COMPUTATIONAL PATTERNS AND BENCHMARKS

Dwarf	**Sample Benchmarks**	R_{SDC}*
Dense Linear Algebra	LINPACK [37], BLAS [38]	15% - 25% [7]
Sparse Linear Algebra	SparseBLAS, SPARSKIT	15% - 25% [7]
Spectral Methods	FFT [39], NAS FT [40]	9% - 12%
N-Body Methods	CoMD [41], Lulesh [35]	30% - 35%
Structured Grids	NAS MG, LU [40]	15% - 25% [7]
Unstructured Grids	UGAWG** [42]	15% - 20%
Map Reduce	Stencil, Monte	1% - 5% [7]
Combinational Logic	Hashing, SPEC 2006 CRC, RSA	25% - 37%
Graph Traversal	Rodinia BFS [33]	10% [7]
Dynamic Programming	SPEC2006 Integer: Go (gobmk)	30%
Backtrack and Branch-and-Bound	SAT solvers, Knuth's Dancing Links	6% [7]
Finite State Machines	SPEC2006 Integer: Text processing (perlbench)	23%
Graphical Models	BUGS, Infer.NET	10% [7]

*It is the range of the SDC rate tested on benchmarks.
**These benchmarks have been re-written in C language in our work.

the "critical" instructions is essential. Different computational patterns have varying SDC rates and rank differently in terms of criticality. By identifying the "dwarfs" sections, we can rank these sections and select the most "critical" ones. Here, we introduce LLMs in our task and present our implementation and results below.

With the continuous development of techniques, large language models (LLMs) have achieved significant progress across various tasks. When software developers use these LLMs as programming assistants, their performance leads users to believe that the models comprehend program semantics well. However, the question remains: can these LLMs understand the semantic aspects of computational patterns, particularly the "thirteen dwarfs of parallelism," which categorize common computational tasks in parallel applications? Some previous works have confirmed that these LLMs can understand code syntax, code static semantic structures, and code dynamic behaviors through diverse tasks [43]. It is unclear whether these LLMs can comprehend the resilience of loops based on their semantic characteristics, particularly their association with the dwarfs. Moreover, if LLMs can identify computational patterns, the extent to which they comprehend the semantics is also unknown.

To address these questions, we conduct a progressive analysis to explore the capability of LLMs in comprehending program semantics in terms of computational patterns. We use the GPT-4o [1], the latest flagship model. GPT-4o ("o" for "omni") is a step towards much more natural human-computer interaction—it accepts any combination of text, audio, image, and video inputs, and generates any combination of text, audio, and image outputs. It can respond to audio inputs in as little as 232 milliseconds, with an average of 320 milliseconds, which is similar to human response time in a conversation. It matches GPT-4 Turbo performance on text in English and code.

[1]https://openai.com/index/hello-gpt-4o/

GPT-based models utilize the prompt-based learning paradigm. The design of the prompt can significantly impact the performance of the response. To design better prompts, we drafted multiple versions and manually evaluated the responses to observe their differences. Based on the experience obtained from these trials, we developed a summarized prompt template known as the role prompt. The role prompt assigns a specific role to LLMs, providing a task context for the model to effectively generate the desired output. The template is shown below:

:

You are [ROLE] familiar with [Domain Knowledge]. [Task Description]. [OUTPUT Format]. The input is [INPUT].

In this template, the placeholder [ROLE] specifies the particular role assigned to the LLMs, such as "HPC expert." The [Domain Knowledge] section defines the domain expertise required for the task, for example, "the 13 dwarfs." The [Task Description] outlines the expected task for the LLMs to perform, providing clear guidance on what is required. The [Output Format] section details the output specification, ensuring the response is structured appropriately. Finally, the [INPUT] denotes the programming language used for the code analysis, which can be either source code or Intermediate Representation (IR) code compiled from LLVM.

In our study, we utilized this role-based prompting approach to address RQ3. The final prompt used is as follows:

:

You are an HPC expert familiar with the 13 dwarfs/computational patterns. Analyze the input code to identify the computational

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

Analysis of the Code:

1. Initialization and Random Assignment (Structured Grids)

```
void initialize_spins(int spins[L][L]) {
    for (int i = 0; i < L; i++) {
        for (int j = 0; j < L; j++) {
            spins[i][j] = (rand() % 2) * 2 - 1;  // Randomly assign +1 or
-1      }
    }
}
```

2. Energy Computation (Structured Grids)

```
double compute_energy(int spins[L][L]) {
    double energy = 0.0;
    for (int i = 0; i < L; i++) {
        for (int j = 0; j < L; j++) {
            int right = (i + 1) % L;
            int down = (j + 1) % L;
            energy -= spins[i][j] * (spins[right][j] + spins[i]
[down]);}
    }
    return energy;
}
```

3. Monte Carlo Step (N-Body Methods and Structured Grids)

```
void monte_carlo_step(int spins[L][L], double temperature) {
    for (int n = 0; n < N; n++) {
        int i = rand() % L;
        int j = rand() % L;

        int right = (i + 1) % L;
        int left = (i - 1 + L) % L;
        int down = (j + 1) % L;
        int up = (j - 1 + L) % L;

        int dE = 2 * spins[i][j] * (spins[right][j] + spins[left][j] + spins[i][down] + spins[i]
[up]);
        if (dE <= 0 || ((double)rand() / RAND_MAX) < exp(-dE / temperature)) {
            spins[i][j] = -spins[i][j];
        }
    }
}
```

This function performs Monte Carlo steps, where each step involves selecting a random spin and potentially flipping it based on its neighbors. The interaction computation (dE) can be seen as an **N-Body Methods** pattern, while the regular 2D grid traversal fits into the **Structured Grids** pattern

Fig. 1. The results from ChatGPT with our prompt

patterns used. Provide the patterns identified and specify the corresponding parts of the code.

This approach led to more accurate and relevant responses, improving the overall quality of the semantic analysis conducted by the LLM.

We then implemented this prompt on a dataset of source code and IR code. We manually observed each pattern identified with the corresponding parts of the code. Impressively, the results were perfect, as shown in Figure 1. This demonstrates that using LLMs to identify the dwarfs and observe methods to pinpoint the ”dwarfs” sections allows us to develop future methods, such as instruction duplication, to protect application execution.

VI. Discussion and Future Work

Our study provides insights into the resilience of loops in HPC programs by analyzing the susceptibility of various computational patterns, known as dwarfs, to soft errors, particularly Silent Data Corruptions (SDCs). However, several limitations must be acknowledged. Firstly, our method evaluates SDC rates using entire applications rather than specific loop sections. This limitation implies that the resilience characteristics derived may not fully represent the exact behavior of the loops.

Furthermore, while our use of Large Language Models (LLMs) demonstrated promising results in identifying computational patterns, the accuracy and reliability of LLMs in selecting critical sections for resilience enhancement remain uncertain. The dependency on LLMs introduces potential biases and limitations, necessitating further validation to ensure their effectiveness in real-world scenarios.

Future research should focus on narrowing the scope of fault injection experiments to test the resilience of specific loops. This will enable a more precise understanding of resilience across different loops. Additionally, future work should aim to optimize instruction duplication techniques by leveraging the loop semantic understanding of LLMs. Selectively duplicating critical instructions or code sections based on their resilience characteristics can reduce overhead while maintaining high levels of fault tolerance.

VII. Conclusion

This paper addresses the critical need for resilience in High-Performance Computing (HPC) applications by focusing on the vulnerability of program loops to transient hardware faults. We systematically identified and summarized the loop patterns corresponding to the thirteen dwarfs of parallelism, revealing the intricate relationship between computational patterns and resilience. Through extensive fault injection experiments, we quantified the Silent Data Corruption (SDC) rates associated with these dwarfs, providing valuable insights into their susceptibility to soft errors. Furthermore, our innovative use of Large Language Models (LLMs) with prompt engineering has demonstrated a practical approach to identifying these dwarfs within real source code. Our findings underscore the importance of targeted resilience strategies for the most critical loops, enhancing the overall reliability and performance of HPC applications. Future work will explore the optimization of duplication techniques and the application of our methods to a broader range of HPC systems and applications.

Acknowledgment

Thanks to the support partially from NSF #2217104 #2212465.

REFERENCES

[1] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and materials reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[2] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[3] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pp. 1–10, 2013.

[4] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen, "Fault tolerant matrix-matrix multiplication: correcting soft errors on-line," in *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, 2011, pp. 25–28.

[5] G. B. Joseph Sloan and R. Kumar, "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance on parallel systems," in *DSN*, 2013, pp. 1–12.

[6] H. Jiang, J. Zhu, B. Fang, C. Chen, R. Jin, and Q. Guan, "Happa: A modular platform for hpc application resilience analysis with llms embedded," *Proceedings of the 43rd International Symposium on Reliable Distributed Systems (SRDS)*, 2024.

[7] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: a methodology for evaluating the error resilience of gpgpu applications," in *Performance Analysis of Systems and Software (ISPASS), 2014*, 2014.

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[9] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[10] R. A. Poldrack, T. Lu, and G. Beguš, "Ai-assisted coding: Experiments with gpt-4," *arXiv preprint arXiv:2304.13187*, 2023.

[11] T. Kadosh, N. Hasabnis, V. A. Vo, N. Schneider, N. Krien, A. Wasay, N. Ahmed, T. Willke, G. Tamir, Y. Pinter *et al.*, "Scope is all you need: Transforming llms for hpc code," *arXiv preprint arXiv:2308.09440*, 2023.

[12] M. Shen, B. Jiang, J. Y. Zhang, and O. Koyejo, "Batch active learning from the perspective of sparse approximation," in *2022 Conference on Neural Information Processing Systems (NeurIPS) Workshop on Human in the Loop Learning.*, 2022.

[13] X. Ding, L. Chen, M. Emani, C. Liao, P.-H. Lin, T. Vanderbruggen, Z. Xie, A. Cerpa, and W. Du, "Hpc-gpt: Integrating large language model for high-performance computing," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 951–960.

[14] B. Jiang, Z. Zhuang, S. S. Shivakumar, D. Roth, and C. J. Taylor, "Multi-agent vqa: Exploring multi-agent foundation models in zero-shot visual question answering," in *The IEEE/CVF Conference on Computer Vision and Pattern Recognition 2024 Workshop on What is Next in Multimodal Foundation Models?*, 2024.

[15] Y. Xie, B. Jiang, T. Mallick, J. D. Bergerson, J. K. Hutchison, D. R. Verner, J. Branham, M. R. Alexander, R. B. Ross, Y. Feng, L.-A. Levy *et al.*, "Wildfiregpt: Tailored large language model for wildfire analysis," *arXiv preprint arXiv:2402.07877*, 2024.

[16] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, "Modeling parallel programs using large language models," *arXiv preprint arXiv:2306.17281*, 2023.

[17] N. K. A. Le Chen, A. Dutta *et al.*, "The landscape and challenges of hpc research and llms," *arXiv preprint arxiv:2402.02018*, 2024.

[18] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, E. T. Barr, and W. Le, "A comprehensive study of the capabilities of large language models for vulnerability detection," *arXiv preprint arXiv:2403.17218*, 2024.

[19] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[20] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," in *2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2017, pp. 97–108.

[21] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "epvf: An enhanced program vulnerability factor methodology for cross-layer resilience analysis," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), acceptance rate = 21%*, June 2016, pp. 168–179.

[22] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 27–38.

[23] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic soft error reliability on the cheap," *SIGPLAN Not.*, vol. 45, no. 3, Mar.

[24] J. Calhoun, L. Olson, and M. Snir, "Flipit: An llvm based fault injector for hpc," in *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I 20*. Springer, 2014, pp. 547–558.

[25] Z. Li, H. Menon, K. Mohror, P.-T. Bremer, Y. Livant, and V. Pascucci, "Understanding a program's resiliency through error propagation," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 362–373.

[26] G. Georgakoudis, I. Laguna, D. S. Nikolopoulos, and M. Schulz, "Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–14.

[27] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2017, pp. 249–258.

[28] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[29] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 375–382.

[30] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.

[31] Q. Guan, X. Hu, T. Grove, B. Fang, H. Jiang, H. Yin, and N. DeBadeleben, "Chaser: An enhanced fault injection tool for tracing soft errors in mpi applications," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 355–363.

[32] H. Jiang, S. Ruan, B. Fang, Y. Wang, and Q. Guan, "Visilience: An interactive visualization framework for resilience analysis using control-flow graph," in *2023 IEEE 28th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2023, pp. 250–256.

[33] 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 *IISWC*, ser. IISWC '09.

[34] N. P. Benchmarks, "Nas parallel benchmarks," *CG and IS*, 2006.

[35] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards,

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *IEEE IPDPS 2013*, Boston, USA.
[36] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *DSN*, June 2014.
[37] J. J. Dongarra, "The linpack benchmark: An explanation," in *International Conference on Supercomputing*. Springer, 1987, pp. 456–474.
[38] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *2010 IEEE computer society annual symposium on VLSI*. IEEE, 2010, pp. 288–293.
[39] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," *ACM SIGARCH computer architecture news*, vol. 23, no. 2, pp. 24–36, 1995.
[40] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.
[41] P. Cicotti, S. M. Mniszewski, and L. Carrington, "An evaluation of threaded models for a classical md proxy application," in *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, Nov.
[42] D. Ibanez, N. Barral, J. Krakos, A. Loseille, T. Michal, and M. Park, "First benchmark of the unstructured grid adaptation working group," *Procedia engineering*, vol. 203, pp. 154–166, 2017.
[43] W. Ma, S. Liu, Z. Lin, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, L. Li, and Y. Liu, "Lms: Understanding code syntax and semantics for code analysis," *arXiv preprint arXiv:2305.12138*, 2023.
[44] T. A. Davis, "Algorithm 1000: Suitesparse: Graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.
[45] X. Liang, J. Chen, D. Tao, S. Li, P. Wu, H. Li, K. Ouyang, Y. Liu, F. Song, and Z. Chen, "Correcting soft errors online in fast fourier transform," ser. SC, 2017.
[46] E. Lindahl, B. Hess, and D. van der Spoel, "Gromacs 3.0: a package for molecular simulation and trajectory analysis," *Journal of Molecular Modeling*, vol. 7, no. 8, p. 306–317, Aug. 2001. [Online]. Available: http://dx.doi.org/10.1007/s008940100045
[47] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. In't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen *et al.*, "Lammps-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, 2022.

APPENDIX

In this Appendix, we provide definitions of computational dwarfs, their loop characteristics, sample code, and associated benchmarks.

1) Dense Linear Algebra:

- **Definition:** Operations involving dense matrices, such as matrix multiplications and factorizations.
- **Loop Characteristics:** Nested loops iterating on matrix dimensions.
- **Sample Code:**

```
// Matrix multiplication
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        C[i][j] = 0;
        for (int k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

- **Benchmarks:** LINPACK [37], BLAS [38]

2) Sparse Linear Algebra:

- **Definition:** Operations involving sparse matrices, such as sparse matrix-vector multiplication and iterative solvers.
- **Loop Characteristics:** Iteration over nonzero elements, often using compressed storage formats.
- **Sample Code:**

```
// Sparse matrix-vector multiplication
for (int i = 0; i < n; i++) {
    y[i] = 0;
    for (int j = row_ptr[i]; j < row_ptr[i
        ↪ +1]; j++) {
        y[i] += values[j] * x[col_idx[j]];
    }
}
```

- **Benchmarks:** SuiteSparse [44]

3) Spectral Methods:

- **Definition:** Methods that involve transformations such as Fast Fourier Transforms (FFT).
- **Loop Characteristics:** Often single loop or simple nested loops, with FFT being the primary operation.
- **Sample Code:**

```
// FFT operation
fftw_plan plan = fftw_plan_dft_1d(n, in, out,
    ↪ FFTW_FORWARD,
    ↪ FFTW_ESTIMATE);
fftw_execute(plan);
fftw_destroy_plan(plan);
```

- **Benchmarks:** FFTW [45], NAS Parallel Benchmarks [34]

4) N-Body Methods:

- **Definition:** Simulations where interactions between a large number of particles are computed, typically involving gravitational or electrostatic forces.
- **Loop Characteristics:** Double nested loops for pairwise interactions.
- **Sample Code:**

```
// N-body simulation
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (i != j) {
            double dx = x[j] - x[i];
            double dy = y[j] - y[i];
            double dist = sqrt(dx*dx + dy*dy
                ↪ );
            force[i] += G * mass[i] * mass[j]
                ↪ / (dist * dist);
        }
    }
}
```

- **Benchmarks:** GROMACS [46], LAMMPS [47]

5) Structured Grids:

- **Definition:** Problems where computation is performed on a regular grid, such as finite difference methods for solving partial differential equations.

- **Loop Characteristics:** Double or triple nested loops iterating over regular grid points.
- **Sample Code:**

```
// Finite difference method for solving heat equation
for (int i = 1; i < n-1; i++) {
    for (int j = 1; j < n-1; j++) {
        u_new[i][j] = u[i][j] + alpha * (u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j]);
    }
}
```

- **Benchmarks:** NAS Parallel Benchmarks (LU, MG) [34]

6) Unstructured Grids:

- **Definition:** Similar to structured grids but with irregular connectivity, often used in finite element analysis.
- **Loop Characteristics:** Nested loops iterating over elements with irregular connectivity.
- **Sample Code:**

```
// Finite element method
for (int elem = 0; elem < num_elems; elem++) {
    double Ke[4][4]; // Element stiffness matrix
    assemble_element_stiffness_matrix(Ke, elem);
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            K[global_nodes[elem][i]][global_nodes[elem][j]] += Ke[i][j];
        }
    }
}
```

- **Benchmarks:** UGAWG Cone-cone, Cube [42]

7) Map Reduce:

- **Definition:** Distributed processing of large data sets, involving map and reduce operations.
- **Loop Characteristics:** Parallelizable loops with independent computations.
- **Sample Code:**

```
// Map function
for (int i = 0; i < n; i++) {
    map_output[i] = map_function(data[i]);
}
// Reduce function
for (int i = 0; i < n; i++) {
    reduce_output = reduce_function(reduce_output, map_output[i]);
}
```

- **Benchmarks:** Stencil, Monte

8) Combinational Logic:

- **Definition:** Combinational Logic refers to the type of logic circuit whose output is a pure function of the present input only. It involves logic gates performing bitwise operations, logic operations, and simple arithmetic operations, often in straightforward, un-nested loops.
- **Loop Characteristics:** Un-nested loops focused on bitwise operations, logical operations (AND, OR, XOR, NOT), or simple arithmetic (addition, subtraction). These loops typically iterate over array elements or bits within integers.
- **Sample Code:**

```
// Bitwise AND operation on two arrays
for (int i = 0; i < n; i++) {
    C[i] = A[i] & B[i];
}
// XOR operation to check parity
for (int i = 0; i < n; i++) {
    if ((A[i] ^ B[i]) == 0) {
        // Do something
    }
}
// Simple arithmetic operation: adding two arrays
for (int i = 0; i < n; i++) {
    C[i] = A[i] + B[i];
}
```

- **Benchmarks:** Hashing, SPEC 2006 CRC, RSA

9) Graph Traversal:

- **Definition:** Operations on graphs, including search algorithms like depth-first search (DFS) and breadth-first search (BFS).
- **Loop Characteristics:** Loops iterating over nodes and edges, often using recursion or queues/stacks.
- **Sample Code:**

```
// Depth-First Search (DFS)
void DFS(int v, bool visited []) {
    visited[v] = true;
    for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i, visited);
}
```

- **Benchmarks:** Rodinia BFS [33]

10) Dynamic Programming:

- **Definition:** Method for solving complex problems by breaking them down into simpler subproblems, storing the results of subproblems to avoid redundant computations.
- **Loop Characteristics:** Loops filling a table based on the results of previous computations.
- **Sample Code:**

```
// Fibonacci sequence
int fib[n+1];
fib[0] = 0;
fib[1] = 1;
for (int i = 2; i <= n; i++) {
    fib[i] = fib[i-1] + fib[i-2];
}
```

- **Benchmarks:** SPEC2006 Integer: Go (gobmk)

11) Backtrack and Branch-and-Bound:

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

- **Definition:** Algorithmic techniques for solving combinatorial optimization problems by exploring possible solutions incrementally and abandoning those that do not meet the criteria.
- **Loop Characteristics:** Recursion with loops trying out possible steps.
- **Sample Code:**

```
    // N-Queens problem
bool solveNQueens(int board[N][N], int col) {
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQueens(board, col + 1))
                return true;
            board[i][col] = 0;
        }
    }
    return false;
}
```

- **Benchmarks:** SAT solvers, Knuth's Dancing Links

12) Finite State Machines:

- **Definition:** Computations based on state transitions, often used in control systems and protocol design.
- **Loop Characteristics:** Loops involving state transitions based on conditions.
- **Sample Code:**

```
    // Simple finite state machine
enum State {STATE_A, STATE_B, STATE_C};
State currentState = STATE_A;
while (true) {
    switch (currentState) {
        case STATE_A:
            if (condition) currentState =
                ↪ STATE_B;
            break;
        case STATE_B:
            if (condition) currentState =
                ↪ STATE_C;
            break;
        case STATE_C:
            if (condition) currentState =
                ↪ STATE_A;
            break;
    }
}
```

- **Benchmarks:** SPEC2006 Integer: Text processing (perlbench)

13) Graphical Models:

- **Definition:** Probabilistic models representing complex dependencies among variables, often used in machine learning and statistics.
- **Loop Characteristics:** Iterative loops refining probabilistic estimations.
- **Sample Code:**

```
    // Simple belief propagation
for (int iter = 0; iter < max_iters; iter++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            messages[i][j] = computeMessage(i, j)
                ↪ ;
        }
    }
    for (int i = 0; i < n; i++) {
        beliefs[i] = computeBelief(i, messages);
    }
}
```

- **Benchmarks:** BUGS, Infer.NET

979-8-3503-8713-1/24/$31.00 ©2024 IEEE