Quantitatively Modeling Application Resilience with the Data Vulnerability Factor

Li Yu*, Dong Li†, Sparsh Mittal†, and Jeffrey S. Vetter†§

†Oak Ridge National Laboratory, *Illinois Institute of Technology,
§Georgia Institute of Technology
{lyu17}@iit.edu, {lid1, mittals, vetter}@ornl.gov

***Abstract*—Recent strategies to improve the observable resilience of applications require the ability to classify vulnerabilities of individual components (e.g., data structures, instructions) of an application, and then, selectively apply protection mechanisms to its critical components. To facilitate this vulnerability classification, it is important to have accurate, quantitative techniques that can be applied uniformly and automatically across real-world applications. Traditional methods cannot effectively quantify vulnerability, because they lack a holistic view to examine system resilience, and come with prohibitive evaluation costs. In this paper, we introduce a data-driven, practical methodology to analyze these application vulnerabilities using a novel resilience metric: the data vulnerability factor (DVF). DVF integrates knowledge from both the application and target hardware into the calculation. To calculate DVF, we extend a performance modeling language to provide a structured, fast modeling solution. We evaluate our methodology on six representative computational kernels; we demonstrate the significance of DVF by quantifying the impact of algorithm optimization on vulnerability, and by quantifying the effectiveness of specific hardware protection mechanisms.**

I. Introduction

The continued growth of today's extreme-scale systems is fueled by the two trends: continued integration of additional functionality onto system nodes, and the increased number of nodes (and components) in the systems [7]. Meanwhile, succeeding generations of these systems introduce new challenges in managing system performance, power, and reliability.

In this regard, strategies for improving the visible resilience of extreme-scale applications continue to evolve. Recent techniques for resiliency [13] require the ability to estimate the resilience of each specific application component rather than treating the application processes monolithically [8], [19], [22], [29]. This work extends our previous work [24] that revealed a wide range of responses of application data structures to system memory errors.

Resilience variance within an application demands a quantitative approach to measure application vulnerability with enough resolution to distinguish the resilience differences across application components in order to selectively apply protection mechanisms to those important components. We believe that selective use of these safeguards is critical when balancing their benefits against the costs of their respective overheads. Hence, quantifying real-world application resilience uniformly and automatically is absolutely essential for practical design of resilient extreme-scale systems.

When an application is executed on a specific hardware, the visible resilience of the application is determined by both application and hardware. In particular, the application algorithm and implementation impact the sensitivity of the application to errors [9], [10], [20] and constrain error propagation [32], [41]; the hardware decides fault patterns (i.e., how frequently and where the faults happen), which in turn impacts runtime states of the application. Hence, an accurate resilience study of extreme-scale applications should use a holistic view and capture the effects of both application and hardware.

The existing methodologies to understand application resilience are not sufficient to guide the resilient system design. They either rely on statistical-based fault injection or detailed architecture analysis. The statistical-based fault injection technique injects random faults into application states [10], [24], [41] or hardware components [23], [33]. To ensure statistical significance, researchers have to perform a large amount of fault injection operations, which is prohibitively expensive. In addition, researchers can only statistically characterize application resilience; there is no capability to quantitatively compare the resilience of application components, which limits the application of fault injection results to optimize resilience mechanisms. The other existing methodology, the detailed architecture analysis [6], [31], is highly hardware-oriented, and it cannot perform fined-grained resilience analysis (e.g., at the granularity of data structures) at the application level.

In this paper, we introduce an analytical model-based approach to quantify application resilience. We propose a novel resilience metric, named the *data vulnerability factor* (DVF), to quantify vulnerability of individual data structures. The creation of DVF intends to holistically examine the system stack and capture the impacts of both application and hardware on data vulnerability. Hence, the introduction of DVF seeks to avoid the isolation between application and hardware when evaluating application resilience. In this paper, we limit our study to a specific hardware component, the main memory. But the definition of DVF is also applicable to other hardware components (e.g., cache hierarchy, register file and network interface card).

In this paper, we use a data-driven approach and focus on quantifying the resilience of data structures (e.g., the tree structure in the Barnes-Hut N-body simulation and the matrices in matrix multiplication). Using the data-driven approach is

978-1-4799-5500-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SC.2014.62

critical for the resilience research, because HPC applications are characterized with a large amount of data structures, and the application outputs are typically stored in data structures.

Furthermore, many popular resilience mechanisms are designed to protect data structures (e.g., checkpointing and specific algorithm-based fault tolerance methods [11], [12], [14], [15], [17]). Quantifying the resilience of data structures can benefit the designs of those mechanisms and implement selective protection with minimal overhead.

To measure DVF, we introduce a novel modeling method based on a domain specific language (DSL) for system modeling, Aspen [35]. We extend the syntax and semantics of Aspen to facilitate the DVF calculation. Furthermore, we categorize memory access patterns after investigating a number of HPC application kernels, and derive a set of general approaches to calculate DVF. Based on the extended Aspen, researchers can quickly explore and compare the resilience of data structures with various hardware options; researchers can also investigate the trade-off between performance and resilience, shown in our two use cases. Furthermore, using the Aspen-based approach, the evaluation cost is at the time granularity of seconds, much smaller than the evaluation costs associated with the statistical-based fault injection and detailed architecture analysis.

The major contributions of this paper are summarized as follows:

1) We create an analytical model-based approach to analyze application resilience based on a novel resilience metric, DVF; DVF integrates the resilience effects of both application and specific hardware into the resilience analysis, hence providing a more complete view of system resilience than the traditional methods;
2) We introduce a method to measure DVF by extending a domain specific language for system modeling. This method provides a fast solution to model application resilience;
3) We measure six numerical kernels from a spectrum of computational domains, including dense linear algebra, sparse linear algebra, N-body methods, structured grids, spectral methods and Monte Carlo;
4) We demonstrate the values of DVF by two use cases. In the first use case, we quantify the impact of algorithm optimization on algorithm resilience; in the second use case, we quantify the effectiveness of a hardware protection mechanism in terms of resilience and explore the tradeoff between performance and resilience. The two use cases show that the DVF-based analytical model provides valuable guides to the design of algorithms and optimization of hardware-based protection.

II. Introduction to Aspen

Our resilience modeling is based on Aspen. Aspen is a domain specific language for structured analytical modeling of applications and architectures. Aspen specifies a formal grammar to describe an abstract machine model and describe an application's behaviors, including available parallelism, operation counts, data structures, and control flow. Aspen's DSL constraints models to fit the formal language specification, which enforces similar concepts across models and allows for correctness checks. Aspen is designed to enable rapid exploration of new algorithm and architectures. Because of the succinctness, expressiveness and composability of Aspen, we use Aspen as the vehicle for our resilience modeling. A more detailed description of Aspen can be found from [35].

Based on Aspen, our resilience modeling intends to capture resilience effects of both application and hardware. Furthermore, our resilience modeling is designed to be a facilitator for coarse-grained exploration of application resilience on specific hardware. Like prior Aspen, our resilience modeling intends to achieve excellent flexibility, performance, ease and scalability with balanced accuracy. It complements traditional simulation approaches by avoiding detailed architectural information and detailed application source code.

In this paper, we model application resilience with the consideration of a single hardware component, main memory. Our ongoing work involves additional hardware components and critical application behaviors into the resilience modeling.

III. Resilience Modeling

In this section, we discuss the resilience modeling in more details. We first introduce the concept of DVF. Then we explain how to measure DVF based on Aspen. The notations for the resilience modeling are summarized in Table I.

TABLE I. Notations for the resiliency modeling

DVF_d	DVF for a specific data structure
FIT	Failure rate (i.e., failures per billion hours per Mbit)
T	Application execution time
S_d	Size of data structure
N_{error}	Number of errors that could occur to a specific data structure during application execution
N_{ha}	Number of accesses to hardware (the main memory in this work)
n	Number of major data structures in an application
DVF_a	DVF for the application

A. DVF: A New Resilience Metric

DVF is designed to quantify the effects of hardware and application on application resilience. DVF for a specific data structure (DVF_d) is defined in Equation (1). Generally speaking, DVF for a data structure is defined as the multiplication of number of errors (N_{error}) and number of accesses to the hardware component due to accesses to the data structure (N_{ha}). N_{error} refers to the errors that could happen to the data structure due to hardware failure. Given the execution time, failure rate and data size, N_{error} can be calculated as $N_{error} = FIT * T * S_d$. We discuss how to calculate N_{ha} in Section III-B.

$$\begin{aligned} DVF_d &= N_{error} * N_{ha} \\ &= FIT * T * S_d * N_{ha} \end{aligned} \tag{1}$$

The term N_{error} captures critical hardware effects (i.e., the failure rate) and application effects (i.e., the execution time and data size); The term N_{ha} captures application effects (i.e., the memory access pattern). Intuitively, a higher hardware failure rate, a longer execution time, a larger data size, and a larger

number of accesses to a data structure indicate that the data structure is more vulnerable, because the data structure has more chances to be corrupted and the corruption has more chances to affect the application. The definition of DVF for a data structure is aligned with the above intuition: a larger value of DVF indicates a more vulnerable data structure and vice versa.

The definition of DVF for a data structure employs a straight multiplication of N_{error} and N_{ha}, based on an implicit assumption that these two resilience effects have equal contributions to the vulnerability of the data structure. In fact, many other metrics (e.g., the energy which is the product of power and time, EDP which is the product of energy and delay, and the impulse which is the product of force and time), use the similar assumption to capture the effects of multiple contributing factors. A further refined definition of DVF could assign a weighting factor to each term to account for diverse vulnerability contributions from each term.

DVF for an application (DVF_a) is defined in Equation (2). Generally speaking, DVF_a is the numerical summation of DVFs of those major data structures within the application. The combination of major data structures accounts for most of the working set size of an application, and most of the computational operations happen to those data structures. Hence, DVF_a is employed to evaluate the application vulnerability.

$$DVF_a = \sum_{i=1}^{n} DVF_{d_i} \tag{2}$$

Based on the above definition of DVF, we are able to perform comparative studies of resilience at the level of fine-grained data structure and coarse-grained application. In addition, we can use DVF in a number of research scenarios. For example, we use DVF to evaluate the algorithm optimization and quickly explore the tradeoff between performance and resilience (shown in Section V); we use DVF to decide whether a specific resilience mechanism provides sufficient protection, given a pre-defined DVF target; we use DVF to compare the effectiveness of diverse fault tolerance mechanisms; we use DVF to determine if a data structure is vulnerable and whether we should enforce extra protection.

To calculate DVF, we must calculate N_{ha}. We discuss it in the next two subsections. The general workflow of calculating DVF with Aspen is described in Section III-D.

B. Counting Main Memory Accesses Based on Data Access Patterns

Counting the number of main memory accesses to a data structure is challenging because of the following reasons. First, we must consider the caching effects. The cache hierarchy reduces the number of accesses to the main memory. The caching effects are tightly coupled with memory access patterns which are determined by the application. Second, we intend to maintain the successful paradigm of Aspen which demands no detailed application source code and limited architectural information while providing fast exploration of application characteristics on various hardware options. However, this paradigm imposes a great challenge on counting memory accesses, because accurately counting the number of memory accesses heavily depends on the details of application implementation and system designs. Third, we perform the memory access analysis at the granularity of individual data structures. This is different from the traditional methods that focus on the whole application working set. Analyzing memory access at the data structure level requires establishing connections between data semantics and memory accesses. This further imposes challenges on our analysis.

We introduce a novel analytical modeling method to address the above challenges, namely coarse grained, pseudocode-based memory access accounting (CGPMAC). CGPMAC works at the high-level pseudo code, making it independent of the application implementation details. CGPMAC leverages the access order of critical data structures extracted from the high-level pseudo code, and estimates the number of main memory accesses due to the last level cache misses and evictions, based on the probability analysis and the coarse grained reuse analysis. Hence, CGPMAC intends to capture temporal access patterns and provide a strong indication to the realistic memory accesses.

After investigating a number of representative HPC application kernels, we classify memory access patterns into four classes to facilitate CGPMAC-based analysis. The memory access pattern for a specific data structure can be characterized by the composition of these four classes of memory access patterns. Hence, our method inherits modularity and composability of Aspen. We will discuss these four classes and describe how CGPMAC is applied to the six numerical algorithms listed in Table II.

C. Four Classes of Data Access Patterns

The four generalized memory access patterns are streaming access, random access, template-based access, and data reuse access. We explain how to estimate the number of main memory accesses corresponding to each of these memory access patterns. In addition, we only consider the last level cache during analysis, because it has the largest impact on the number of main memory accesses within the cache hierarchy. This is especially true for inclusive caches. The parameters for the last level cache and data structures used in our models are summarized as below.

TABLE III. NOTATIONS FOR THE LAST LEVEL CACHE AND DATA STRUCTURES.

C_c	Cache capacity
C_A	Cache associativity
N_A	Number of cache sets
C_L	Cache line length
D	Data structure size
N	Number of elements in a data structure
E	Size of a single element

Streaming Access Pattern The streaming access is defined as a sequential traverse of a data structure with a fixed stride length. Since each element in the data structure is accessed at most once, all the main memory accesses are caused by compulsory cache misses. Figure 1 shows an example, in which S is the access stride length. To estimate the number of main memory accesses for the streaming access pattern, we consider three cases.

TABLE II. SIX NUMERICAL ALGORITHMS EMPLOYED IN THIS WORK

Algorithm name	Computational method class	Major data structures	Memory access patterns	Example benchmarks
Vector Multiplication (VM)	Dense linear algebra	A, B, and C	Streaming	Homemade code
Conjugate Gradient (CG)	Sparse linear algebra	A, x, p and r	Template+Reuse+streaming	NPB CG [2]
Barnes-Hut simulation (NB)	N-body method	T and P	Random	[1]
Multi-grid (MG)	Structured grids	R	Template-based	NPB MG [2]
1D FFT (FT)	Spectral methods	A	Template-based	NPB FT [2]
Monte Carlo simulation (MC)	Monte Carlo	G and E	Random	XSBench [4]

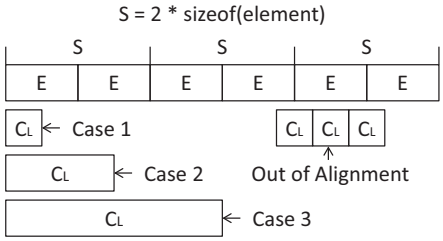


Fig. 1. An example of steaming accesses to a data structure.

In the first case, the cache line length is no larger than the element size (i.e., $C_L \leq E$). Depending on whether the element is aligned with the cache line, each reference to the element either introduces $\lceil E/C_L \rceil$ or $\lceil E/C_L \rceil + 1$ memory accesses. The former occurs when the element is aligned with cache lines, while the later occurs when it is out of alignment. Assuming that each byte within a cache line has the same probability to save the element , then the probability of having $\lceil E/C_L \rceil + 1$ memory accesses (i.e., nonalignment) is calculated as

$$p = \frac{(E-1) \mod C_L}{C_L} \tag{3}$$

Consequently, the expected number of main memory accesses for each element reference (A_E) is

$$\begin{aligned} A_E &= \lceil E/C_L \rceil * (1-p) + (\lceil E/C_L \rceil + 1) * p \\ &= \lceil E/C_L \rceil + p \end{aligned} \tag{4}$$

If the stride length is larger than the element size (i.e., $E < S$), then no cache line can be used by more than one element. The number of elements accessed is $\lfloor D/S \rfloor$, and the number of main memory accesses is estimated as $\lfloor D/S \rfloor * A_E$. If the stride length is equal to the element size (note that the stride length is typically no smaller than the element size), all cache lines used by the data structure need to be loaded. The total number of memory access is estimated as $\lceil D/C_L \rceil$.

In the second case, the cache line length is between the element size and the stride length (i.e., $E < C_L \leq S$). Similar to the previous case, each reference to an element in this case introduces either 1 or 2 memory accesses, depending on whether the element is aligned with the cache line. The expected number of main memory accesses for each element reference is $1 * (1-p) + 2 * p = 1 + p$, and the number of elements accessed is $\lfloor D/S \rfloor$. Hence the total number of main memory access is estimated as $\lfloor D/S \rfloor * (1+p)$.

In the third case, the cache line length is larger than the stride length (i.e., $S < C_L$). All the cache lines used by the data structure need to be loaded and the total number of main memory access is estimated as $\lceil D/C_L \rceil$.

Random Access Pattern The random access pattern exists in a number of scientific applications such as N-body simulation and Monte Carlo simulation. This pattern is characterized with a computation loop within which the elements of the target data structure are randomly accessed in each iteration, and whether there is any access to each element depends on control flows and runtime states. For example, the Barnes-Hut algorithm (N-body simulation) organizes n bodies (i.e., the elements) into a quad-tree (i.e. the data structure). To calculate the net force on a particular body, the tree is traversed to calculate the force acting on the body. During the tree traverse, whether a particular body will be accessed or not depends on the physics, which is random and dependent on the runtime states. To calculate the net force for all bodies, the tree must be repeatedly traversed. Hence, unlike the steaming access pattern, the random access could result in an unpredictable number of accesses to each element. During the modeling, we assume that each element in the target data structure is already traversed once before the random accesses happen. This assumption is used to model the data construction phase, commonly found in many scientific applications.

Our modeling for the random access is based on a probability analysis. In particular, we calculate the probability that the data elements are still in the cache after random visits, and then we estimate the number of data blocks that need to be loaded into the cache based on the calculated probability. To analyze the random access pattern for a target data structure, we require five parameters as inputs, including (1) the number of elements in the target data structure, denoted by N; (2) the element size denoted by E; (3) the average number of distinct elements visited in each iteration denoted by k; (4) the number of iterations denoted by $iter$; and (5) the ratio of cache blocks occupied by the target data structure to the whole cache blocks, denoted by r.

The parameters k and $iter$ are used to reduce randomness for modeling and make the analysis of memory access formalizable. These two parameters are usually output as a part of the application results, hence they can be easily obtained by profiling application on any available hardware. The parameter r is used to model the cache interferences between concurrent random accesses to multiple data structures. In particular, we model the impact of the cache interference by dividing the cache among all data structures. Each data structure gets only a fraction of the cache according to its size. For example, for the Monte Carlo simulation, two data structures (Grid and Energy) are randomly and concurrently accessed. Given the data sizes S_{grid} and S_{eng} for the Grid and Energy respectively, the Grid

gets the cache size $S_{grid}/(S_{grid}+S_{eng})*C_c$ while the Energy gets the cache size $S_{eng}/(S_{grid}+S_{eng})*C_c$. The number of main memory accesses for each data structure is estimated based on a fraction of the cache instead of the whole cache. In fact, a similar method has been employed to model the cache interferences in [28].

Depending on the relationship between the data structure size and the cache capacity, we categorize the random access into two cases.

In the first case, all of the data elements can be loaded into the cache (i.e., $E*N \leq C_c*r$). The random accesses to the data structure only introduce compulsory cache misses. The number of memory access is estimated as $\lceil E*N/C_L \rceil$.

In the second case, the cache capacity is smaller than the size of the data structure (i.e., $C_c*r < E*N$). The first accesses to the data elements still causes $\lceil E*N/C_L \rceil$ times memory access. However when some of the elements are randomly re-visited, they may not be in the cache and cause additional memory accesses. To calculate the number of main memory accesses for this case, we first calculate the expected number of elements that are not in the cache, then we calculate the number of cache blocks that must be loaded in order to load these missed elements.

Let X be a variable representing the number of elements that are not in the cache when k distinct elements are visited, we have

$$P(X=x) = \frac{\binom{k}{k-x}\binom{N-k}{m-k+x}}{\binom{N}{m}} \tag{5}$$

where m is the number of elements that can be loaded into the cache (i.e., $m = C_c*r/E$) at the same time. Equation 5 describes the possibility of any $(k-x)$ elements out of k distinct elements appearing in any m elements loaded into the cache. Based on Equation 5, we can calculate the expected number of elements not in the cache, shown in Equation 6.

$$X_E = \sum_{x=1}^{min\{N-m,k\}} P(X=x)*x \tag{6}$$

Given X_E, we now calculate the number of cache blocks that need to be read from the main memory in order to load X_E elements. In particular, if the element size is larger than the cache line size (i.e., $C_L < E$), then the number of needed cache blocks is roughly calculated as $B_{elm} = \lceil (E/C_L) \rceil * X_E$; otherwise, the number of needed cache blocks is roughly estimated as $B_{elm} = X_E$, which is the largest possible number of needed cache blocks (the number of needed cache blocks could be smaller). Furthermore, as the total number of cache blocks used by the data structure is $E*N/C_L$ and C_A*N_A*r cache blocks are in the cache, the number of cache blocks not in the cache is $B_{out} = E*N/C_L - C_A*N_A*r$. This implies that the X_E elements cannot be loaded by more than B_{out} cache blocks.

Based on the above discussion, we estimate the average number of cache blocks that need to be reloaded per iteration as

$$B_{reload} = min(B_{elm}, B_{out}) \tag{7}$$

Finally, the total number of main memory accesses for $iter$ iterations plus the data initialization is calculated as $\lceil E*N/C_L \rceil + B_{reload}*iter$ (note that $\lceil E*N/C_L \rceil$ comes from the initial accesses to the target data structure before random accesses).

Template-Based Access Pattern Some data structures have very complex memory access patterns falling between the streaming access and the random access. In particular, these access patterns are more complicated than the streaming access but cannot be simply categorized into the random access because they follow specific rules. For example, for the mesh-based PDE solver, the accesses to mesh elements of the data structure *mesh* follow specific topology or stencil information instead of arbitrarily constructed. We call this kind of access pattern, the *template-based access pattern*.

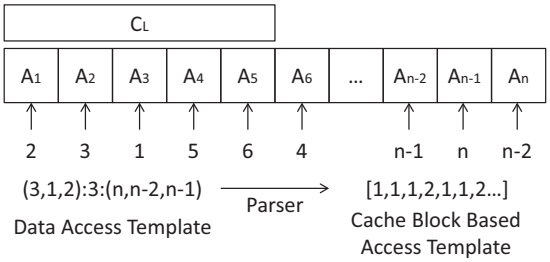


Fig. 2. An example of the template based access.

To estimate the number of main memory accesses for a data structure with the template-based access pattern, the users are required to input detailed information about the memory access. Assume the size of the data structure is D and the number of cache blocks needed to load the data structure is $n = \lceil D/C_L \rceil$. We represent the data structure by a set of data blocks $B = \{b_1, b_2, ..., b_n\}$. Then the access template T can be any combination of the elements in B. Section III-D gives an example of how to express the template. According to the memory access template, we estimate the number of main memory accesses by the following two-steps algorithm.

- Step 1. For each data block b_i in template T, if b_i appears for the first time, we increase the number of the memory accesses by one.
- Step 2. If b_i appears more than once, then we calculate the distance between this appearance and the immediate last appearance, denoted by d. If d is larger than the maximum available cache capacity, then a cache miss could occur. We then increase the number of main memory accesses by one.

In addition, to improve the usability of this model, we allow users to input the template based on data structure elements, and the template can be expressed in a regular expression similar to the one in Matlab. The parser then converts the template to a cache block-based template. Figure 2 shows an example.

Data Reuse Pattern The data reuse pattern indicates that data is repeatedly accessed by the application. When the

data is reused, the requested data may or may not be in the cache because of the cache interference from the other existing data structures. The random access pattern and the template-based access pattern also have the characteristics of data reuse. However, the data reuse in the random access pattern is unpredictable; the data reuse in the template-based access pattern is highly regular and structural. The data reuse pattern in this section refers to those cases not included in the random and template-based access patterns. In particular, this pattern is predictable, but it cannot be easily expressed with a template. Also, this pattern requires a thorough consideration of the cache interferences between multiple data structures. For example, the data structure p_k in CG (see Algorithm 4) is repeatedly used within each iteration, however we cannot easily use a template to describe the reuse pattern because of the complex program structure and logic. Also, the accesses to p_k are interfered by other data structures such as A, x and r. In the following discussion, we will use the terms cache block and data block interchangeably.

Similar to the random access model, the key idea to model the data reuse pattern is based on a probability analysis. We first model that the target data structure is exclusively loaded into the cache (Equation 8) or concurrently loaded into the cache with other interfering data structures (Equation 10); after the target data structure is loaded, we model how the target data structure is reused and interleaved with the other interfering data structures (Equations 11 and 12).

Similar to [38], we model the allocation of data blocks into cache associative sets as a Bernoulli trial (Equation 8), i.e., a data block has equal opportunity to be allocated into any of the associative sets. We define that A is the target data structure with a size F_A in terms of the number of cache blocks and X_A is a variable representing the number of cache blocks left by A in a single cache set when A exclusively uses the cache. The probability that x cache blocks are left by A in a cache set is estimated as

$$P(X_A = x) = \begin{cases} (\frac{1}{N_A})^x * (1 - \frac{1}{N_A})^{F_A - x}, & x < C_A \\ \sum_{i=A}^{F_A} (\frac{1}{N_A})^i * (1 - \frac{1}{N_A})^{F_A - i}, & x = C_A \end{cases} \tag{8}$$

where $\frac{1}{N_A}$ is the probability that a data block of A falls into one of the cache sets and the cache associativity C_A is the maximum number of cache blocks that can be left in one set.

The expected number of cache blocks left by A in a cache set is

$$E(X_A) = \sum_{r=1}^{C_A} P(X_A = r) * r \tag{9}$$

Now we consider the scenario when A is loaded concurrently with other data structures. To simply the model, we consider other data structures as a whole, denoted by B with F_B as its size in terms of the number of cache blocks. X_B is a variable representing the number of cache blocks left by B in a cache set when B exclusively use the cache. R_A is a variable representing the number of cache blocks left by A in a cache set when A and B are loaded concurrently. We have

$$P(R_A = r | X_A = x, X_B = y) = \begin{cases} 1, r = x \ and \ x + y \leq C_A \\ 1, r = C_A * \frac{x}{x+y} \ and \ x + y > C_A \\ 0, otherwise \end{cases} \tag{10}$$

Equation 10 states that: (1) if the cache blocks left by A and B for the exclusive usage of the cache can be loaded into the cache at the same time, then there is no interference; (2) otherwise, the cache blocks allocated to A are a fraction (i.e., $x/(x+y)$) of a cache set.

After A is accessed and loaded into the cache, the interfering data B can be immediately accessed and impact the future reuse of A. There are two possible scenarios. The first scenario follows Equation 8 (i.e., A is exclusively loaded); the second scenario follows Equation 10 (i.e., A is concurrently loaded with other data structures). For the first scenario, if the cache replace happens, the accesses to B will replace cache blocks not belonging to A because of the LRU policy and because A is just accessed. After all the other cache blocks in each set have been replaced, the cache blocks for A will be replaced. Based on the above analysis, we calculate the possibility that A leaves r data blocks in a cache set after accessing B as

$$P(R_A = r | X_A = x, X_B = y) = \begin{cases} 1, x + y \leq C_A \ and \ r = x \\ 1, x + y > C_A \ and \ r = C_A - y \\ 0, otherwise \end{cases} \tag{11}$$

Equation 11 states that if there is no interference (i.e., $x+y \leq C_A$) then all the cache blocks for A will stay in the cache; otherwise, a specific number $(x+y-C_A)$ of cache blocks for A will be replaced.

For the second scenario, we first calculate the expected number of data blocks left by A and B based on Equations 8 and 9. This can be done by regarding A and B as a combined, single data structure. We refer this expected number as I. If the cache replace happens, any of I cache blocks can be replaced. We calculate the possibility that A leaves r data blocks in a cache set after accessing B as

$$P(R_A = r | X_A = x, X_B = y) = \frac{\binom{x}{x-r}\binom{I-x}{y-x+r}}{\binom{I}{y}} \tag{12}$$

Based on Equations 8 and 10-12, we calculate $P(R_A = r)$ as follows.

$$\begin{aligned} &P(R_A = r, X_A = x, X_B = y) \\ &= P(R_A = r | X_A = x, X_B = y) \\ &* P(X_A = x) * P(X_B = y) \end{aligned} \tag{13}$$

$$P(R_A = r) = \sum_{x=0}^{C_A} \sum_{y=0}^{C_A} P(R_A = r, X_A = x, X_B = y) \quad (14)$$

Based on Equation 14, we calculate the expected number of cache blocks left by A in a cache set as

$$E(R_A) = \sum_{r=1}^{C_A} P(R_A = r) * r \quad (15)$$

The number of main memory accesses to A includes loading those elements not in the cache, which is estimated as $F_A - N_A * E(R_A)$.

D. Extending Aspen

We extend syntax and semantics of Aspen to allow users to express memory access patterns of a target data structure and describe hardware information. We also extend the Aspen compiler to implement the models in the last subsections. Figure 3 depicts the workflow to calculate DVF based on the user input information and Aspen compiler.

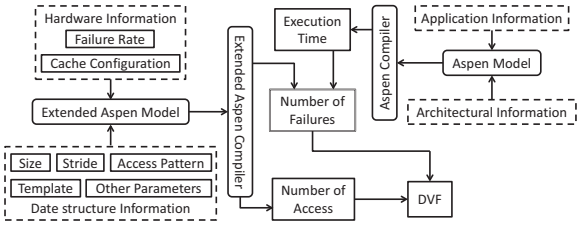


Fig. 3. The workflow to calculate DVF.

To explain the new syntax of Aspen for the memory access patterns, we use four out of the six algorithms (see Table II) as examples. The first algorithm is the vector multiplication (VM) that calculates the product of two arrays, shown in Algorithm 1. In VM, the data structure A, B and C have the stream access patterns, but with different access strides. To describe the streaming access pattern, there are three Aspen parameters, including the element size in bytes, the number of elements in the target data structure, and the the stride length measured by number of elements. An example for the data structure A is shown as follows. This example shows that the data structure A has 200 elements, each of which is 8 bytes, and the stride is $8 * 4 = 32$ bytes (4 elements).

Algorithm 1 Vector Multiplication

```
procedure VM(A, B, C)                          ▷ C=A*B
    for i ← 1, n do
        C_i ← C_i + A_{i*j} * B_{i*k}
    end for
end procedure
```

Aspen Program
Data structure : {A}
Access Pattern : {s}
Parameters : {(8,200,4)}

Algorithm 2 The core procedure for Barnes-Hut algorithm

```
procedure FORCE UPDATE(p,node)
    if region at node contains 1 particle then
        compute force between the two particles
    else
        if p is distant enough from region then
            compute force between p and region center
        else
            for each subnode of node do
                Force Update (p,subnode)
            end for
        end if
    end if
end procedure
```

Aspen Program
Data structure : {T}
Access Pattern : {r}
Parameters : {(1000,32,200,1000,1.0)}

The second example is the core procedure of *Barnes-Hut* algorithm that is widely used for simulating the N-body problem. In this procedure, all nodes are organized into a tree structure (T), and each node needs to be compared with a portion of other nodes in the tree. The number of comparisons for each node depends on the tree structure and particle's mass which are usually generated randomly. Hence the memory accesses to the tree are random. Algorithm 2 depicts the procedure. As mentioned earlier, the parameters to describe the random access pattern include N, E, k, $iter$ and r. In this example, there are 1000 tree nodes ($N = 1000$) with size 32 bytes ($E = 32$). The number of iteration is 1000 ($iter = 1000$), and the average number of node comparisons in each iteration is 200 ($k = 200$). The cache ratio factor is 1.0 ($r = 1.0$).

Algorithm 3 A smoother procedure in Multi-grid

```
procedure SMOOTHER(R)          ▷ R(i,j,k)=i*n2*n1+j*n1+k
    for i ← 2, n3 − 1 do
        for j ← 2, n2 − 1 do
            for k ← 1, n1 do
                R[i] = R(i,(j-1),k)
                  + R(i,(j+1),k)
                  + R((i-1),j,k)
                  + R((i+1),j,k)
            end for
        end for
    end for
end procedure
```

Aspen Program
Data structure : {R}
Access Pattern : {t}
Parameters : {(16)}
Template : $\{(R(2,1,1), R(2,3,1), R(1,2,1), R(2,2,1)) : 1 : (R(n3-1, n2-2, n1), R(n3-1, n2, n1), R(n3-2, n2-1, n1), R(n3, n2-1, n1))\}$

The third example is for the template-based access pattern. Algorithm 3 shows the smoother procedure in Multi-grid

algorithm (MG), in which the grid (denoted by R) is accessed following a specific template. According to the pseudocode, the procedure starts with sequential references of four starting elements in R (i.e., $R(2,1,1), R(2,3,1), R(1,2,1)$ and $R(2,2,1)$) and then within each iteration the algorithm advances accesses to the elements in the last iteration by one until reaching the grid boundary (i.e., $R(n3-1, n2-2, n1), R(n3-1, n2, n1), R(n3-2, n2-1, n1)$ and $R(n3, n2-1, n1)$). In the template, the four starting elements are expressed as a function of $n1$, $n2$ and $n3$. For example, the first reference element is $R(2,1,1) = 2 * n2 * n1 + n1 + 1$. Also, we input the element size (16) as a parameter, based on which the input template can be converted to the cache block-based template.

The fourth example is a complex example, particularly Conjugate Gradient method (CG), including the reuse pattern and other patterns. Algorithm 4 shows one iteration in the CG, in which four data structures (i.e., A, x, p, r) are referenced and reused. To describe the access patterns, the model input includes a list of data structures (A, r, p, x), a list of data access order based on the pseudo code for reuse analysis, a list of data access patterns for individual data structures, a list of parameter sets to describe data structures (i.e., data structure size, element size, and stride size) for the streaming patterns and templates. In this example, due to the space limit, we do not show the templates for data structure A and p and some parameter sets for p, r, and x.

Algorithm 4 One iteration in Conjugate Gradient method

procedure ITERATION(A, r, p, x)

$\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}$

$x_{k+1} = x_k + \alpha_k p_k$

$r_{k+1} = r_k - \alpha_k A p_k$

$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$

$p_{k+1} = r_{k+1} + \beta_k p_k$

$k \leftarrow k + 1$

end procedure

Aspen Program

Data structure : {A r p x}
Access order : {r(Ap)p(xp)(Ap)r(rp)}
Access Pattern : {s(tt)s(ss)(tt)s(ss)}
Parameters : {(8,200,4)...}
Template : {...}

IV. RESULTS

In this section, we first validate our models by estimating the number of main memory accesses. Then we measure DVF for diverse numeral algorithms (i.e., DVF profiling), and study the implications of DVF on algorithm designs. To validate the memory access models, we develop a tool to collect memory references based on Pin [27]. We also develop a configurable cache simulator that uses the memory references as input and counts number of memory accesses for specific data structures after caching. We then compare the number of memory accesses reported by the cache simulator with the number estimated by our models. The cache simulation is based on the popular LRU algorithm and can report the number of cache misses and writebacks. We simulate a last level cache during the model verification. The cache configurations are summarized in Table IV. During the verification and DVF profiling, we focus on the major computation parts of the algorithms, and ignore initialization and finalization phases, because most computation operations happen in those computation parts. The FT algorithm used in our experiments is a segment of codes from the NPB FT benchmark that conducts a 1D FFT computation. For the MG algorithm, we use the V-cycle kernel and skip the other segments.

TABLE IV. CACHE CONFIGURATION.

Cache	C_A	N_A	C_L	C_c
Small (Verification)	4	64	32 bytes	8KB
Large (Verification)	16	4096	64 bytes	4MB
16KB (Profiling)	2	1024	8 bytes	16KB
128KB (Profiling)	4	2048	16 bytes	128KM
1MB (Profiling)	6	4096	32 bytes	1MB
8MB (Profiling)	8	8192	64 bytes	8MB

A. Verification of Estimating Number of Main Memory Accesses

For the model verification, we choose two sets of cache configurations (i.e., the small and large ones shown in Table IV). We also use a set of relatively small input sizes (Table V) for the algorithms, because the cache simulation is very time consuming with the memory traces of the large input problem sizes. For the small cache configuration, we choose the cache to be small enough to have cache interferences. Figure 4 presents the verification results.

The figure shows that our model provides accurate estimations for all access patterns in general. The estimation error is within 15% in all cases. For the random access pattern (Barnes-hut and Monte Carlo), even though there is access randomness, given the sufficient information from users, our model still achieves good estimation accuracy.

TABLE V. APPLICATION INPUT SIZE (VERIFICATION).

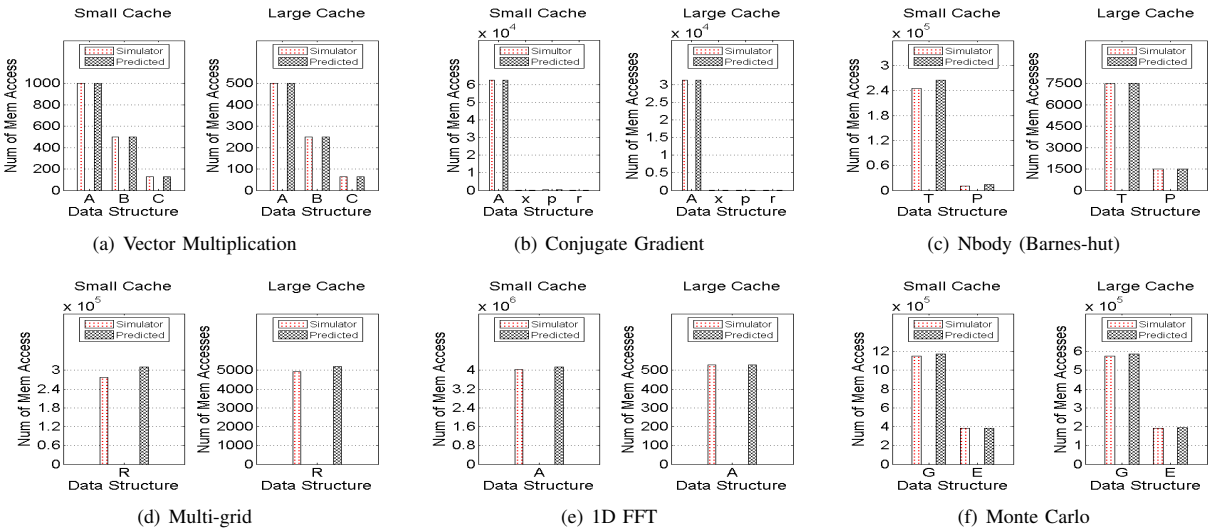
Application	Input size
VM	10^3 Integer Array
CG	500*500 Double Matrix [3]
NB	1000 Particles [1]
MG	Problem class = S [2]
FT	Problem class = S [2]
MC	Size = small, Lookups = 10^3 [4]

B. DVF Profiling

TABLE VI. APPLICATION INPUT SIZE (PROFILING).

Application	Input size
VM	10^5 Integer Array
CG	800*800 Double Matrix [3]
NB	6000 Particles [1]
MG	Problem class = W [2]
FT	Problem class = S [2]
MC	Size = small, Lookups = 10^5 [4]

For DVF profiling, we can use a set of relatively large input sizes summarized in Table VI, because of the low evaluation cost of our modeling method. We choose four cache configurations with diverse configurations shown in Table IV to study the sensitivity of DVF to the cache configurations. The results are presented in Figure 5.



(a) Vector Multiplication (b) Conjugate Gradient (c) Nbody (Barnes-hut)

(d) Multi-grid (e) 1D FFT (f) Monte Carlo

Fig. 4. Model Verification

We first notice that different data structures within the same application can have different DVFs. An interesting observation is from the VM algorithm (See Figure 5(a)), in which the data structure A has obviously larger DVF than the data structures B and C. Because the execution times for the three data structures are the same, the difference in DVF mainly comes from the footprint size of the data structure and the number of main memory accesses. In our experiments, the data structure A has a larger stride length than B and C; both the footprint size and the number of main accesses of A are larger than B and C. This observation tells us that the data access pattern can affect DVF significantly.

We further compare DVFs across algorithms. We notice that CG and FT are both memory-intensive algorithms, but the DVF for our CG implementation can be thousands of times larger than that for the FT implementation (See Figure 5(b) and 5(e)). We find that this big difference mainly comes from two reasons. First, the working set size of CG in our implementation is more than 100 times larger than that of FT (i.e., 5000KB vs. 33KB). Second, the execution time of CG is more than 200 times longer than that of FT. Although the number of main memory accesses for FT is larger than that for CG, the larger working set size and longer execution time of the CG implementation makes it more vulnerable than that of the FT implementation.

A similar observation comes from NB and MC (See Figure 5(c) and 5(f)). Both applications have the random access pattern but the DVF for MC is much larger than that for NB. We find that the DVF difference is partially caused by the larger working set size and the longer execution time of MC. In addition, although the average number of node comparisons in MC is smaller than that in NB (i.e., 1 and 80), the number of iterations in MC is larger (10^5 and 6000), which leads to a bigger number of main memory accesses of MC than NB.

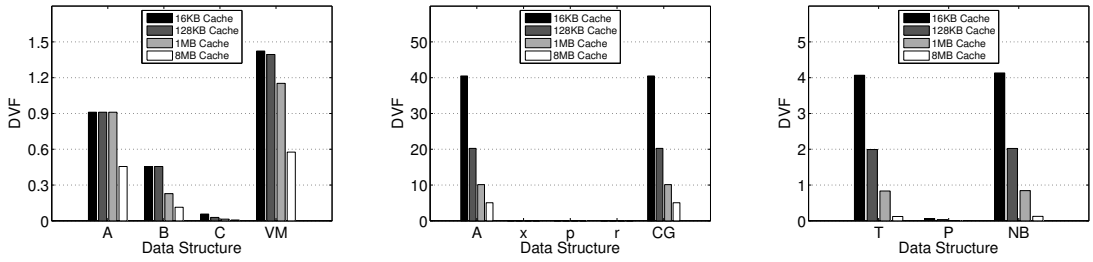
In addition, we find that the algorithms show different sensitivities to the cache capacity. For instance, the DVF values for the FT algorithm increase suddenly when the cache capacity is smaller than a threshold (i.e., 16KB in Figure 5(e)). This sudden change is caused by the specific access pattern of the FT algorithm. In this access pattern, the same data structure is traversed multiple times following a specific template. If the cache cannot load the entire data structure, there is a large number of cache misses, thus leading to an increase of the number of main memory accesses. Other algorithms do not show such a sudden change of DVF because they have diverse access patterns. The streaming access pattern does not have the sudden change of DVF because the data structure is traversed only once and the number of main memory accesses remains relatively stable across cache configurations. For the random access pattern, when the cache capacity is not enough for the whole data structure, the DVF increases gradually (not suddenly).

V. Use Cases

In this section, we demonstrate the significance of our resilience modeling work with two use cases: an example of algorithm optimization that examines the impact of algorithm-level optimization on application resilience; and an example of hardware protection that evaluates the tradeoff between performance and resilience. The cache configuration used in this section is the largest cache in Table IV.

A. Quantifying the Impact of Algorithm Optimization on Vulnerability

The algorithm optimization traditionally aims to improve performance and energy efficiency, but it is widely unknown how can a specific algorithm optimization impact the algorithm resilience. To implement scalable and efficient execution of applications on the extreme-scale systems, it is critical to extend the algorithm study into multiple dimensions (i.e., performance, energy and resilience). We particularly study the CG in this section. The regular CG is depicted in Algorithm 4. The preconditioned CG (PCG), as an algorithm optimization (see Algorithm 5), is designed to ensure faster convergence



(a) Vector Multiplication (b) Conjugate Gradient (c) Nbody (Barnes-hut)

(d) Multi-grid (e) 1D FFT (f) Monte Carlo

Fig. 5. DVF Profiling

Algorithm 5 Preconditioned Conjugate Gradient

$r_0 = b - Ax_0$
$z_0 = M^{-1}r_0$
$p_0 = z_0$
$k = 0$
procedure REPEAT(A, M, r, z, p, x)
  $\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k - \alpha_k A p_k$
  if r_{k+1} is sufficiently small **then**
    exit loop
  end if
  $z_{k+1} = M^{-1} z_{k+1}$
  $\beta_k = \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
  $p_{k+1} = z_{k+1} + \beta_k p_k$
  $k = k + 1$
end procedure

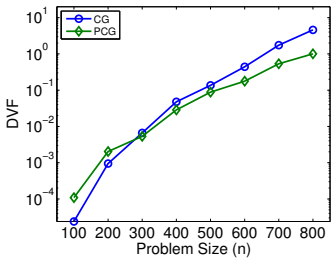


Fig. 6. CG vs PCG.

of CG, and aims to provide better performance. Most of the data structures in PCG are the same as those in CG except an auxiliary matrix M and an auxiliary vector z in PCG. We calculate DVF for CG and PCG, and compare their resilience; we also vary the input problem size and observe how the algorithm vulnerability is correlated with it. Figure 6 shows the results.

The results show that PCG is more vulnerable than CG (but pretty close) with the small input problem sizes (100 and 200); however PCG becomes better than CG with the large input problem sizes. This resilience variance comes from the contradicting contributions of performance improvement and larger working set size to DVF in PCG: On one hand, PCG provides better performance which should reduce DVF; on the other hand, the increase of the working set size increases DVF. In our study, with the small input problem sizes, the performance of PCG is close to CG but PCG uses a larger working set and has more main memory accesses, hence the DVF of PCG becomes worse. However, with the large input problem sizes, the performance benefit of PCG is significantly pronounced and outweighs the negative impact of the larger working set, thus improving DVF. Based on this study, we are able to derive an appropriate input problem size to achieve joint optimization of performance and resilience when applying PCG.

B. Quantifying Effectiveness of A Data Protection Mechanism

The hardware-based resilience mechanisms are commonly employed in high-end computing systems. With the traditional methods for vulnerability study, it is difficult to quantify the effectiveness of those mechanisms in terms of resilience. With the introduction of DVF, we can quickly evaluate them and explore the tradeoff between performance and resilience. In this section, we particularly investigate hardware error checking and correction code (ECC) for main memory. The hardware ECC can correct specific faults occuring in the memory devices, but they also result in performance loss. Chipkill [16] and SECDED [21] are two commonly used ECC. The error rates with these two ECC applied in the main

memory are summarized in Table VII.

TABLE VII. ERROR RATE WITH ECC IN PLACE (FIT=FAILURES PER BILLION HOURS)

ECC Protection	Error Rate (FIT/Mbit)
No ECC	5000 [25], [26]
Chipkill correct	0.02 [26], [34]
SECDED	1300 [26], [39]

Based on Table VII, we can calculate DVF taking into consideration the effect of ECC protection. Figure 7 depicts the variance of DVF with a range of possible performance degradations when applying ECC. From the figure, we notice that DVF is decreased, demonstrating the effectiveness of ECC protection. Furthermore, we notice that DVF achieves the smallest value when the performance degradation is about 5%. A further increase of performance loss (larger than 5%) results in a increase of application vulnerability. This is because a longer execution time due to performance loss makes application more easily hit by the hardware failure, hence reducing the application resilience. Although we do not have real performance results with ECC protection due to hardware limitation, based on Figure 7 we can quickly explore what performance target the ECC mechanism should aim at in order to maximize application resilience.

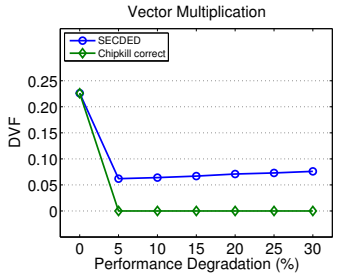


Fig. 7. The impact of ECC on DVF.

VI. RELATED WORK

To understand application vulnerability, the statistical-based random fault injection is one of the major methods. Li et al. [24] build a binary instrumentation-based fault injection tool and perform random fault injection into the data structures of realistic applications based on the PIN infrastructure. Casa et al. [10] inject faults into each instruction's output based on the LLVL typed byte code, and study the vulnerability of algebraic multi-grid solver. Sastry et al. [20] and Xu et al. [41] aggressively employ static and dynamic program analyses to analyze application fault sites and pick a small subset to perform selective fault injections. Their methods greatly reduce the fault injection space. The random fault injection has large evaluation cost, because a large number of fault injections must be performed to obtain statistically meaningful results. Also, this methodology does not consider hardware effects on resilience. It cannot be used to evaluate hardware resilience mechanisms and cannot be used to quantitatively compare the resilience difference between application components.

Besides the statistical-based random fault injection, another class of resilience analysis employs detailed hardware analysis. Mukherjee et al. [30] define the *architectural vulnerability factor* (AVF) as the probability that a fault in a particular structure will result an error. Biswas et al. [5] show how to compute the AVF of address-based processor structures based on a detailed analysis of architecturally correct execution. However, dynamically measuring AVF requires costly performance modeling and simulation. To accelerate AVF analysis, Walcott et al. [40] and Duan et. al [18] identify strong correlations between AVF values and a small set of processor metrics, based on which a faster estimation of AVF is possible. However, a fine-grained (i.e., data structure) application-level analysis is not possible with such analysis. Based on the prior AVF work, Sridharan and Kaeli [36], [37] introduce a new metric to capture the architecture-level fault masking inherent in a program. However, to calculate AVF, one has to use fault injection or an architectural simulator, which can be very costly for evaluation.

Our work in this paper improves current application resilience analysis from three perspectives. First, our methodology considers the effects of both hardware and software; Second, our methodology allows fine-grained analysis (i.e., at the level of data structure); Third, our methodology provides a much faster solution to model application resilience.

VII. CONCLUSION

In this paper, we introduce a methodology to quantify application resilience based on a performance modeling language. Our method captures both hardware and application factors those impact the application resilience. Our method is applied to a spectrum of numerical algorithms and reveals the resilience variance within an algorithm and across different algorithms. More importantly, our work is based on a novel resilience metric, DVF. We demonstrate the significance of our resilience modeling on various optimization problems.

ACKNOWLEDGMENT

REFERENCES

[1] Barnes-hut Implementation on GitHub. http://github.com/JAChapmanII/barnes-hut, 2010.

[2] NPB Website. https://www.nas.nasa.gov/publications/npb.html, 2012.

[3] Conjugate Gradient Implementation on GitHub. https://github.com/danesh-d/cg/blob/master, 2013.

[4] The Monte Carlo Macroscopic Cross Section Lookup Benchmark. https://github.com/jtramm/XSBench, 2013.

[5] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S.S. Mukherjee, and R. Rangan. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *International Symposium on Computer Architecture (ISCA)*, 2005.

[6] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S.S. Mukherjee, and R. Rangan. Computing Architectural Vulnerability Factors for Address-based Structures. In *International Symposium on Computer Architecture (ISCA)*, 2005.

[7] A. Bland, W. Joubert, D. Maxwell, N. Podhorszki, J. Rogers, G. Shipman, and A. Tharrington. Titan: 20-Petaflop Cray XK6 at Oak Ridge National Laboratory. In J.S. Vetter, editor, *Contemporary High Performance Computing: From Petascale Toward Exascale*, CRC Computational Science Series. Taylor and Francis, 2013.

[8] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, and F. Magniette. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2002.

[9] G. Bronevetsky and B.R. Supinski. Soft Error Vulnerability of Iterative Linear Algebra Methods. In *International Conference on Supercomputing (ICS)*, 2008.

[10] M. Casas, B.R. Supinski, G. Bronevetsky, and M. Schulz. Fault Resilience of the Algebraic Multi-grid Solver. In *International Conference on Supercomputing (ICS)*, 2012.

[11] Z. Chen. Algorithm-Based Recovery for Iterative Methods without Checkpointing. In *The International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2011.

[12] Z. Chen. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. In *ACM SIGPLAN Annual Symposium Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[13] J. Chung, I. Lee, M. Sullivan, J.H. Ryoo, D.W. Kim, D.H. Yoon, L. Kaplan, and M. Erez. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2012.

[14] T. Davies and Z. Chen. Correcting Soft Errors Online in LU Factorization. In *The International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2013.

[15] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing. In *International Conference on Supercomputing (ICS)*, 2011.

[16] T. Dell. A White Paper On The Benefits Of Chipkill-Correct ECC for PC Server Main Memory. Technical report, IBM Microelectronics Division, 1997.

[17] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *ACM SIGPLAN Annual Symposium Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[18] L. Duan, B. Li, and L. Peng. Versatile Prediction and Fast Estimation of Architectural Vulnerability Factor from Processor Performance Metrics. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[19] P.H. Hargrove and J.C. Duell. Berkeley Lab Checkpoint/Restart(BLCR) for Linux Clusters. *JPCS*, 2006.

[20] S.K.S. Hari, S.V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *The International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[21] M.Y. Hsiao. A Class of Optimal Minimum Odd-Weight-Column SECDED Codes. *IBM Journal of Research and Development*, 1970.

[22] J. Hursey, J.M. Squyres, T.I. Mattox, and A. Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2007.

[23] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical Fault Injection: Quantified Error and Confidence. In *Design, Automation and Test in Europe (DATE)*, 2009.

[24] D. Li, J.S. Vetter, and W. Yu. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2012.

[25] S. Li, K. Chen, M.-Y. Hsieh, N. Muralimanohar, C.D. Kersey, J.B. Brockman, A.F. Rodrigues, and N.P. Jouppi. System Implications of Memory Reliability in Exascale Computing. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2011.

[26] X. Li, M.C. Huang, K. Shen, and L. Chu. A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility. In *USENIX Annual Technical Conference (ATC)*, 2010.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 2005.

[28] S. Manegold, P. Boncz, and M.L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *International Conference on Very Large Databases (VLDB)*, 2002.

[29] A. Moody, G. Bronevetsky, K. Mohror, and B.R. Supinski. Design, Modeling, and Evaluation of A Scalable Multi-level Checkpointing System. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2010.

[30] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *The Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[31] S.S. Mukherjee, C.T. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. Measuring Architectural Vulnerability Factors. *IEEE Micro*, 2003.

[32] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Characterizing the Impact of Soft Errors on Iterative Methods in Scientific Computing. In *International Conference on Supercomputing (ICS)*, 2011.

[33] G. Shi, J. Enos, M. Showerman, and V. Kindratenko. On Testing GPU Memory for Hard and Soft Errors. In *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, 2009.

[34] C. Slayman. Impact of Error Correction Code and Dynamic Memory Reconfiguration on High-Reliability/Low-Cost Server Memory. In *Integrated Reliability Workshop*, 2006.

[35] K. Spafford and J.S. Vetter. Aspen: A Domain Specific Language for Performance Modeling. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2012.

[36] V. Sridharan and D.R. Kaeli. Eliminating Microarchitectural Dependency From Architectural Vulnerability. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[37] V. Sridharan and D.R. Kaeli. Using PVF Traces to Accelerate AVF Modeling. In *Workshop on Silicon Errors in Logic - System Effects*, 2010.

[38] D. Thiebaut and H.S. Stone. Footprints in the Cache. *ACM Trans. Comput. Syst.*, 1987.

[39] A.N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N.P. Jouppi. LOT-ECC: Localized and Tiered Reliability Mechanisms for Commodity Memory Systems. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[40] K.R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic Prediction of Architectural Vulnerability from Microarchitectural State. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[41] X. Xu and M.-L. Li. Understanding Soft Error Propagation Using Efficient Vulnerability-Driven Fault Injection. In *The Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.