End-to-End Resilience for HPC Applications

Arash Rezaei[1], Harsh Khetawat[1], Onkar Patil[1], Frank Mueller[1](✉), Paul Hargrove[2], and Eric Roman[2]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA
mueller@cs.ncsu.edu

[2] Lawrence Berkeley National Laboratory, Berkeley, CA 94730, USA
{phhargrove,ERoman}@lbl.gov

Abstract. A plethora of resilience techniques have been investigated to protect application kernels. If, however, such techniques are combined and they interact across kernels, new vulnerability windows are created. This work contributes the idea of end-to-end resilience by protecting windows of vulnerability **between** kernels guarded by different resilience techniques. It introduces the live vulnerability factor (LVF), a new metric that quantifies any lack of end-to-end protection for a given data structure. The work further promotes end-to-end application protection across kernels via a pragma-based specification for diverse resilience schemes with minimal programming effort. This lifts the data protection burden from application programmers allowing them to focus solely on algorithms and performance while resilience is specified and subsequently embedded into the code through the compiler/library and supported by the runtime system. In experiments with case studies and benchmarks, end-to-end resilience has an overhead over kernel-specific resilience of less than 3% on average and increases protection against bit flips by a factor of three to four.

Keywords: Resilience · Silent data corruption · Pragma programming

1 Introduction

In large-scale parallel systems, faults are not an exception but rather the norm [18,29]. Faults such as bit flips or hardware faults may result in application or operating system failures. Hardware and software techniques have been

This work was supported in part by a subcontract from Lawrence Berkeley National Laboratory and NSF grants 1525609, 1058779, and 0958311. This manuscript has three authors of Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

devised to make such systems more resilient to failures. But future exascale systems are projected to see an increase in the frequency of faults, which would require 20% more circuitry and energy to counter them [33]. However, hardware vendors tend to design and build general-purpose, and not exascale-specific hardware due to manufacturing costs. As a result, the future systems will be likely built with off-the-shelf components while delegating a significant part of the resilience responsibility to the software layer.

The significance of resilience in future HPC systems has been emphasized in prior research, e.g., [33]. In particular, multiple challenges arise in HPC systems from the size (millions of cores) and the programming model (tightly coupled). Intuitively, larger numbers of components result in a higher probability of failures. What's more, a tightly coupled programming model may result in fast fault propagation after just one node has been hit [17]. Hence, resilience is considered a major roadblock on the path to next-generation HPC systems.

In practice, hardware protection is complemented by software resilience. A variety of software techniques exist, such as checkpoint/restart (CR), redundancy, and algorithm-based fault tolerance (ABFT), each with their own benefits and limitations in terms of applicability and cost. CR has high storage overheads and requires backward recovery via re-execution, which limits scalability [16]. Redundancy requires either only extra memory or both extra memory and processing resources, which is costly [17]. ABFT results in low overheads and supports forward execution, but each numerical algorithm has to be customized [12,15,19]. A choice of a low-cost resilience scheme is best made per numerical kernel rather than for an entire application. The composition of different resilience techniques, however, results in a generally overlooked problem: It creates windows of vulnerability. Consider kernel K1 with redundant execution followed by kernel K2 with ABFT protection. K1's result is consumed by K2, yet the result's integrity is no longer checked after K1 has finished. This leaves variables storing K1's result vulnerable until K2 has consumed all of them. In contrast, by protecting both K1 and K2 with redundancy, intermediate and final results can be compared (dual redundancy) or even corrected (triple redundancy with voting).

We introduce *end-to-end resilience* to allow the selection of different low-cost resilience techniques across different application phases. End-to-end resilience composes protection spaces of kernels with disjoint resilience techniques such that windows of vulnerability are avoided. Another problem is that programmers are often forced to clutter numerical methods with tangential resilience concerns making codes hard to maintain. Resilience APIs try to reduce this clutter but cannot eliminate it, e.g., Containment Domains [9], GVR [39], Charm++ [21], etc. Also, transparent resilience techniques, such as BLCR [13], tend to impose much higher overhead than application-specific resilience via CR [26] or ABFT [15]. But the interleaving of algorithmic and resilience concerns makes it hard to maintain such programs. End-to-end resilience is realized elegantly via pragmas at the program level, which provides the benefits of the aspect-oriented programming (AOP) paradigm [22] as it increases modularity by allowing the separation of algorithmic and resilience concerns at no extra cost while still meshing with a variety of execution paradigms and resilience methods.

This work makes the following contributions:

- We identify the vulnerabilities between protected kernels and offer a systematic solution via end-to-end resilience.
- We propose a metric to quantify vulnerability *across* protected kernels.
- We design and implement a resilience pragma to support separation of the resilience aspects from the algorithms to increase portability and modularity imposing minimal programming effort.
- We show that, in contrast to prior work, auto-generated protection provides full end-to-end protection at less than 3% additional time overhead on average.

2 Background

Hardware faults can be persistent or transient. Persistent faults are typically due to aging or operation beyond temperature thresholds. If a persistent faults results in a failure, re-execution will not help, i.e., an HPC job of thousands of processes is rendered useless. Transient hardware errors, also called soft errors, are often due to cosmic radiation. They allow the application to continue execution, albeit with tainted data. Such faults manifest as bit flips in the data in memory or anywhere in the data path (e.g., caches, data bus). Although CPU registers, caches, and main memory are often equipped with ECC, only single bit flips are correctable while double-flips generally are not (by SEC-DED ECC while chipkill can correct some multi-bit errors depending on their device locality).[1] Jaguar's 360TB of DRAM experienced a double bit flip every 24 h [18]. Some soft faults may remain undetectable and may result in so-called Silent Data Corruption (SDC). SDCs may manifest at application completion by producing wrong results or, prior to that, wrong interim results. It is known that SDC rates are orders of magnitude larger than manufacture specifications [27,30,35].

Resilience methods usually compensate for the computation/state loss by performing a backward or forward recovery. Backward recovery recreates an older state of an application through classic rollback recovery methods, such as system-level or application-level checkpoint/restart (CR) [26]. Forward recovery typically handles errors by repairing the affected data structures. A correction procedure is invoked that may recover the intended values from a peer replica (redundant computing) [17], or via Algorithm-Based Fault Tolerance (ABFT) from checksums or solver properties [8,12,15,19,31].

Many HPC applications are comprised of multiple kernels that form a multi-phase pipeline. The above-mentioned methods are resilient to one or multiple types of faults with different overhead. Intuitively, there is no single solution that fits all scenarios while providing the best performance. Thus, a combination of methods enables the selection of the best resilience mechanism per application phase considering factors such as computation time and size of data that needs protection. End-to-end data integrity is a goal explicitly cited in exascale reports [33]. Our end-to-end resilience fills this very gap.

[1] Bit flips in code (instruction bits) create unpredictable outcomes (most of the time segmentation faults or crashes but sometimes also incorrect but legal jumps) and are out of the scope of this work.

3 Assumptions

Our fault model considers soft errors/SDCs that materialize in memory in a fault agnostic manner, i.e., SDCs may occur in unprotected DRAM (no ECC) due to cosmic rays or may result from bit flips in the processor core during calculations, unprotected register files, or caches. Hence, results of (faulty) calculation are subsequently written to memory, which creates an SDC even if memory is protected with ECC/chipkill. This is consistent with findings of past work [30,35] indicating that undetected errors in SECDED ECC-protected DRAM present a problem today, and that some SRAM structures remain unprotected.

On the software side, we assume that the correctness of a *data structure* can be verified (through a *Checker* method) and the stored values can be recovered through a *Recover* method should an inconsistency be detected. Many algorithms commonly used in HPC, such as numeric solvers, have approximation methods based on convergence tests. These convergence tests could be used as the *Checker*. If an algorithm lacks a simple checking method or invariant, the *Checker* can be provided through comparison with a checksum over the data that was computed beforehand and stored in a safe region.[2] The *Recover* method is given by the forward recovery phase in ABFT methods, or simply by restoring a light-weight deduplicated [2] or compressed [20] checkpoint of the data.

We further assume that the computation is (or can be made) idem-potent *with respect to the encapsulated region*, i.e., if globals are changed inside the region, they have to be restored by the recovery method. In other words, if a method/region is called twice in a row, the result would be the same as the inputs (or global variables) remain unmodified by the computation (no side effects).[3] CR and redundant computing already ensure idem-potency since identical state is restored in the former while redundant state exists for the latter. ABFT methods have to be analyzed to ensure that dynamic live ranges are encapsulated by end-to-end resilience, while any other global side effects need to capture/restore globals at region boundaries. Existing solutions to I/O idem-potency are required as well [4]. We can then retry a computation if needed, i.e., when no other recovery methods exist (or if the other recovery methods have failed). Notice that we do allow the side effects of communication inside regions (see Sect. 4). Application kernels, e.g., the ones studied, were found to be already compliant with these constraints, and frameworks have similar constraints, e.g., DAG-based HPC tasks and map-reduce [1,7].

[2] Extra checks are added to guarantee the correctness of data stored in a safe region. A safe region is assumed to neither be subject to bit flips nor data corruption from the application viewpoint—yet, the techniques to make the region safe remain transparent to the programmer. In other words, a safe region is simply one subject to data protection/verification via checking.

[3] Inputs are read from disk and stored in globals or on the heap, but may be recovered by re-reading from disk. Globals are calculated in the program and can only be recovered by re-calculation or ABFT schemes.

4 End-to-End Resilience

Live Vulnerability Factor: We introduce a new metric, the term Live Vulnerability Factor (LVF): $LVF = L_v \times S_v$,
where L_v is the length of the *dynamic* live range of an arbitrary (incl. non-scalar array) data structure/variable v (vulnerability window), and S_v is the space required to hold the related data in memory. Length is measured as wall-clock time from first set to last use (dynamic live range) of a variable during execution.

Protection Across Scopes: Applications are typically composed of phases during which different algorithmic computations are being performed. Intermediate results are created and passed from phase to phase before the final result is generated. Our core idea is to exploit the *dynamic live range* of predominantly non-scalar variables within and across phases, and to immediately perform a correctness check after the last use of any given variable. *Scalar* live range analysis is a well-understood technique employed by compilers during code optimizations, such as register allocation (among others), while *dynamic* live ranges are rarely analyzed but can, in part, be inferred from escape analysis. Figure 1 outlines the idea for our running example, a sequence of two matrix multiplications, enhanced by an extra checksum row and column per matrix for resilience (see Huang et al. [19]). Huang's method provides protection for result matrices C and E *within* a single matmult kernel (arrows on left side) while end-to-end resilience protects all matrices during their *entire live time across kernels* (arrows on right side). If an error strikes during the lifetime of phase-dependent variables, single-kernel protection methods cannot provide any assistance as they are locally constrained to region boundaries. This is precisely where our end-to-end protection comes to the rescue. In fact, Fig. 1 concisely illustrates that ***single-kernel protection misses out on more than half of the lifetime of variables compared to end-to-end protection*** even if 99% of execution time is spent inside the two matmult kernels.

When a live range ends, data is checked for correctness. If correct, no action is taken, otherwise correct values are recovered (if detected as erroneous), or re-computation is performed (if erroneous but direct recovery has failed). The intuition here is to avoid the high overhead of frequent checks (e.g., after every variable redefinition or use inside the live range) while providing a guaranteed end-to-end correctness of the computation.

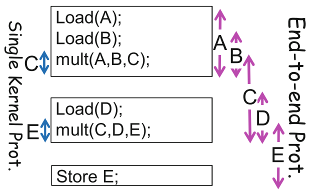

Fig. 1. Matrix multiplication, range of live variables

The Protect Pragma: We propose a pragma-based resilience scheme and show how the corresponding code is expanded to provide the extra end-to-end protection. This allows us to cover the vulnerability window of different variables by automatically expanding codes through the compiler. The expanded code performs check and recovery actions on the vulnerable data. We incorporate end-to-end resilience into OpenMP-like pragmas to facilitate adoption and code maintenance with a potential of future synergy between thread parallelism and resilience (beyond the scope of this paper). The pragma has a simple, yet powerful and extendable interface with the following syntax:

$$\#\textbf{pragma}\ \textit{protect}\ [M][Check(f_1, .., f_n)][Recover(g_1, .., g_m)][Comm][Continue]$$

The resilience method, `M`, which can be `CR` or `Redundancy (2/3)` (dual/triple), is an optional argument. The integration of both resilience approaches is discussed in a latter example. `Check` and `Recover` receive a list of functions parameterized by each variable that needs protection. We use f to denote a checker, and g for a recovery methods. By default, f and g are a checksum calculation and checksum-based correction, respectively. A user may overwrite f/g to supply more efficient checking/recovery. A region that contains MPI communication is annotated with the `Comm` keyword. The `Continue` keyword indicates that data is live beyond the current region, i.e., crossing phases/kernels, and requires end-to-end protection. Figure 2 depicts the source code of our running example with the protect pragmas with the "Continue" keyword to protect live matrices across kernels.

```
Matrix A, B, C, D, E;
Load(A);
Load(B);
#pragma protect Check(Checker(A),Checker(B)) \
                Recover(Correct(A),Load(A),  \
                        Correct(B),Load(B))  \
                 Continue
  mmult(A,B,C); // parallelized
Load(D);
#pragma protect Check(Checker(C),Checker(D)) \
                Recover(Correct(C),Correct(D),Load(D))   \
                Continue
  mmult(C,D,E); // parallelized
#pragma protect Check(Checker(E))
                Recover(Correct(E))
  Store (E);
```

Fig. 2. Matrix multiplication with protect pragma

In the final code, every region is contained within a while loop (protection boundary) with checking and recovery code after the computation. After `mmult(A,B,D)`, a Check is invoked followed by Recover if the check fails inside the loop. (Both are called via function pointers.)

Code resulting from chaining of regions with the `Continue` keyword are highlighted and described as follows. A boolean array of size 3 named `completed` and a flag `first` are maintained for the 3 chained regions in this code, which indicates the correct completion of regions 0, 1, and 2. At the end of region 0/1/2, the corresponding flag is set. Matrix D is only loaded once due to the conditional on the flag. Additional loads may be triggered inside the `Recover()` calls for matrices A, B, and D if they cannot be repaired using checksums.

Recovery from regions that involve MPI communication with other processes requires coordination among these processes. The `Comm` option of the pragma indicates that such communication exists inside that pragma region. It results in generating code for a global reduction of check() return codes indicating if any checks have failed, in which case recovery with recomputation is required where all peer MPI tasks participate in recomputation.

Notice that pragmas cannot easily be replaced by macros. First, variable number of check and recover routines may be specified, one per data structure, which cannot be expressed by a macro. Second, a begin and end macro would be required per pragma, but all three begins would have to be placed on line 4 of Fig. 2 while the ends would follow after lines 8, 13, and 16, respectively. This would make the source code significantly less legible. The compiler also has the ability to perform semantic checks to ensure that the live range of protected variables under the `Continue` keyword extends to the end of the scope spanning multiple pragmas and to capture/restore globals via live range analysis.

Source code changes in terms of added pragmas are only needed at the top level of computations (e.g., timestep loop for most HPC codes), and most notably not in libraries. `Continue` creates an orphaned semantics (in OpenMP terminology) that continues the lexical extent from scope-to-scope (transitively) in the current compilation unit, where each pragma in such a chain expands the live range for the current set of protected variables. For the enclosing scope, the resulting chain results in one checkpoint per pragma in the absence of faults. In the presence of a fault, selective recovery from the last valid data is only performed at the inner-most scope that stored this data.

Task-Based Resilience: An alternative to the pragma approach is to design a task-based programming scheme that implicitly provides end-to-end resilience. Tasking libraries are becoming more popular in the HPC community due to their more graceful load balancing and potentially asynchronous execution models, e.g., PaRSEC [5], OmpSs [14], the UPC Task library [25], and Cilk [3].

Resilience has been added to PaRSEC [6] and OmpSs [24]. Other work focuses on soft faults [6], i.e., they take advantage of the algorithmic properties of ABFT methods to detect and recover from failures at a fine grain (task level) and utilize periodic checkpointing at a coarse grain (application). Yet others uses CR and message logging at the task granularity to tolerate faults with re-execution [24].

Instead of focusing on a specific resilience approach, we target a more complex problem. We propose a tasking system that allows for different resilience methods to interact in an easily understandable and extendable manner. A resilient task class is provided with two methods that are called before and after the actual execution of a task, namely `resilience_pre`, `resilience_post`. In `resilience_pre`, depending on the resilience type of the task, CR or Redundancy, the `checkpoint` method or `wakeup_shadow` is called, respectively. In `resilience_post`, first the shadow process is put to sleep under redundant execution. Then data structures with their last use in the task are checked and corrected if needed. If correction fails, a set of tasks is put into the scheduling queue to recompute the tainted data structures.

5 Implementation Details

The resilience pragma API is implemented as a transform pass in the Cetus compiler [10] via source-to-source transformation (i.e., no need for more complex frameworks like LLVM as the IR is never used). Source-to-source compilation using Cetus allows us to transform an input C program to a modified C program as output. Cetus uses Antlr [28] in order to parse C programs into an Intermediate Representation (IR). The compiler passes are then run on the IR in order to generate the output source code. Each pass iterates over the IR and is capable of modifying it by adding, removing, or editing the input source. New code is added as Cetus IR objects, equivalent to building an IR tree from its leaves. Similarly, a complex IR can be generated by extending these trees. Cetus allows iterating over the IR in a depth-first manner, which is utilized here.

ProtectPragmaParser Class: We added the ProtectPragmaParser class, a transform pass that implements our pragma. Each pragma directive in the input program is represented as an object of the ProtectPragmaParser class. The ProtectPragmaParser class is run in order to transform the generated parse tree to an equivalent parse tree structure, which contains our protection boundaries, checker functionality, and recovery mechanisms. We traverse the input parse tree in a depth-first manner looking for the protect pragma directives. On finding the pragma, we parse the directive to populate the checker and recovery functions associated with this particular pragma. We also generate the necessary protection boundary, checking, and recovery code required in the current context and track the variables defined at these protection boundaries. As part of the ProtectPragmaParser object creation, we check if the current directive is chained to a previously encountered directive via the Continue keyword. If chained, we can recompute these resilient variables in case their recovery methods fail, and the ProtectPragmaParser object of the current context is added to the list of chained pragmas of the directive it is chained to. Otherwise, it is added as an independent (root) pragma. When chaining is found in the input IR, we extend the protection boundaries of the current pragma around that of the following pragma. When the input source code has been completely parsed, a logical structure of these chained (or unchained) directives is created (see pragmas in Fig. 2).

Once the entire input source code has been traversed and the logical structure of pragmas is created, a recursive function that emits transformed code is invoked on the root objects. This, in turn, invokes the function on each of its chained pragmas. It is at this stage that checking and recovery code for non-last-use variables is removed so as to reduce the checking overhead. This function uses the chaining information to correctly emit the nested while loop structure as part of the output source code. As part of the code emitting process, if a particular directive had the `CR` or `Redundancy` clause, then the compiler emits the appropriate function calls to `wake_shadow` and `sleep_shadow` in case of the `Redundancy` clause, and `create_ckpt` in case of the `CR` clause.

The Cetus compiler infrastructure along with our ProtectPragmaParser functionality allows us to transform our input source code in this manner to support end-to-end resilience. While these transformations could be performed manually by the programmer for simple examples, it quickly becomes tedious and error-prone for more complicated program structures or even chained regions. Our Cetus implementation transforms the input source in a single pass through the IR tree, emitting code recursively even for complicated, inter-leaving dependencies between resilient variables. This allows for the development of powerful software that has end-to-end resilience while off-loading the repetitive and sometimes non-trivial task of code expansion to the compiler.

6 Experimental Results

All experiments were conducted on a cluster of 108 nodes, each with two AMD Opteron 6128 processors (16 cores total) and 32GB RAM running CentOS 7.3 and Linux 4.10 (except for TF-IDF, which uses CentOS 5.5, Linux kernel 2.6.32 and Open MPI 1.6.1 due to BLCR [13] and RedMPI [17] requirements). ABFT resilience is realized via protecting critical data with checksums so that we can attempt to recover (repair) results, or, if recovery fails, resort to CR and reload data from disk. Redundancy is realized via Red-MPI of which we obtained a copy [17]. These techniques, referred to as *conventional* resilience in the following, are compared to their equivalent version with end-to-end resilience guarantees.

We present examples of pragma- and task-based end-to-end resilience for two variants of matrix multiplication and a page ranking program, with experimental results for these codes as well as the NAS Parallel benchmark codes. To this end, we already discussed end-to-end resilience for two successive matmult kernels in Fig. 2. The same kernels can also be refactored using fine-grained tasking as discussed next.

The task-based resilience class/capabilities (Sect. 4) plus a task-based runtime system are utilized to implement a blocked matrix multiplication utilizing POSIX threads. We add checksums per block of a matrix. The checksum elements are colored in the 2 examples of Fig. 3.

For a matrix of size 4×4, if the block size k is 2, then 20 extra elements are needed to hold the checksums. For a 6×6 matrix, 45 extra elements are needed. In practice, the size of a block (configured to fit into L1 cache with other data) is much larger than the extra space overhead for checksums.

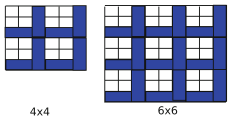

Fig. 3. Blocks (white) with checksums (blue) (Color figure online)

6.1 Matrix Multiplication

We use 5 input sizes for square matrices from 512×512 to 2560×2560. The size of last level cache (L3) is 12 MB, and only the first experiment ($N = 512$) completely fits in the L3 data cache. Thus, data is repeatedly loaded from main memory (DRAM) in all other experiments. We use 16 OpenMP threads that perform matrix multiplications in a blocked manner with a tile/block size of 32×32. Each thread needs 3 blocks to perform the multiplication. Thus, the block size is selected as number of elements that can be accommodated in $\frac{1}{4}$th of the L1 data cache size of 64 KB.

Figure 4 contrasts the performance evaluation of sequentially composed matrix multiplication with conventional resilience (left bar) with our end-to-end resilience (right bar). For the pragma-based solution (left half), fault-free execution ranges from 0.88 ($n = 512$) to 35 s ($n = 2560$) when no correction needs to be triggered. In this case, end-to-end resilience has a 0.99% overhead at $n = 512$; for larger matrix sizes, this overhead is also negligible (around 0.69%). Task-based execution (right half) results in slightly higher execution times and overheads that are between 1.4% (for large matrices) and −0.64% (for small ones) and varies between 4.6% and −6.5% due cache artifacts with unaligned (1536) and aligned (2048) cache lines resulting in more/fewer cache conflicts, respectively. The alignment can be attributed to the implementation of per-block checksums in task-based matrix multiplication. Overall, more computation is performed during the multiplications and check operations.

Observation 1: End-to-end resilience across kernels results in the same cost as conventional resilience only protecting single kernels.

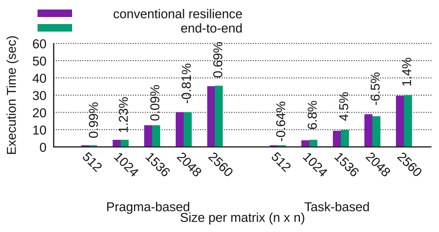

Fig. 4. Time of conventional/end-to-end resilience

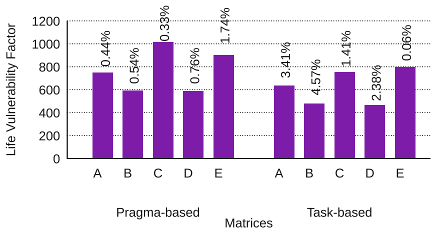

Fig. 5. Live Vulnerability Factor (bars) and % increase (above bars)

Performance Under Faults and Resulting Failures: We next investigate the correlation between LVF and the likelihood of failures in matrices. The LVF is computed from the vulnerability window of data structures (see Sect. 4). Figure 5 depicts the LVF as bars of each matrix under failure-free execution of the application. The vulnerability size is 50.03 MB and the vulnerability window depends on the live range of each matrix. C has the highest LVF, next comes E and then A. B and D have the same LVF, the smallest among the 5 matrices. This reflects the live ranges of the respective (same size) data structures during program execution (see Fig. 1). Notice that conventional resilience would only protect matrices C and E within, but not across kernels, i.e., they would only protect about 50% of our LVF for C/E and none for A/B/D (see Fig. 1). Furthermore, end-to-end resilience adds overhead that increases the LVF by only 0–5%, as depicted by the labels above bars in Fig. 5, but, unlike previous work, checks/corrects SDCs even *across* kernels that are otherwise only locally protected. Since these matrices comprise 99% of the program's data, the LVF per program is the average of weighted per-data LVFs, where weights are 1 (same size and range), except for matrix C with weight 2 (same size, twice the range).

Observation 2: End-to-end resilience protects data over significantly larger execution ranges at less than 1% increased LVF for pragma-based and 2.2% for task-based execution.

We also developed a program variant that injects faults (single bit flips) in uniformly randomized locations over the matrices (all 5 matrices, each sized at 2560×2560) and also at uniformly randomized times in a time window according to a given rate (configurable). This allows us study the effect of fault injections in real life and compare the results to the LVF metric. We randomly inject faults during runtime with fault rates from 25 to 45 s for pragma-based execution. Such high fault rates may be unlikely, but the point is to assess overhead and to illustrate the robustness of our technique: A second fault may be injected before the first one has been mitigated, yet end-to-end resilience is capable of making forward progress. (Solar flares are actually reported to result in multiple SDCs in rapid succession.) The y axis of Fig. 6 shows the number of faults. Using conventional resilience, only the faults in the lower-most shaded region of matrices C and E can be corrected by conventional resilience methods that are limited to a given scope/kernel, such as [11]. For end-to-end resilience, faults resulting in detectable errors in the lower portion of all matrices (errors across and in kernels, i.e., including the shaded regions of C/E) are *all subsequently corrected by end-to-end resilience*, even though they cross scope/kernel boundaries. This is the most significant result of our work as it demonstrates how much more fault coverage end-to-end resilience has compared to conventional resilience schemes. This covers cases where injections hit data while it is *live*. In fact, it shows that the majority of faults occurs *outside* of ABFT kernel protection, which is exactly what end-to-end resilience protects.

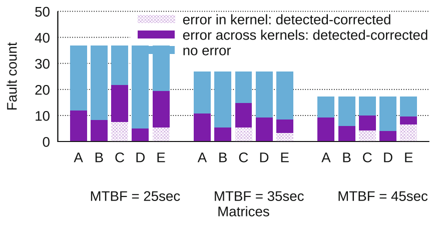

Fig. 6. Fault injection over 100 runs (pragma-based)

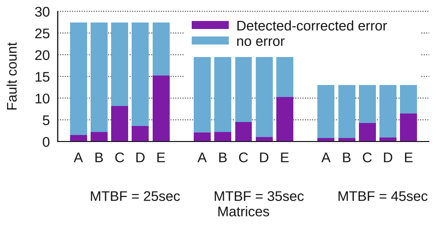

Fig. 7. Fault injection over 100 runs (task-based)

Other injections do not result in a failure as they hit stale data (uppermost portion per bar). In other words, end-to-end resilience never resulted in erroneous results while conventional ABFT misses errors across kernels, which are dominant. Furthermore, the distribution of corrected injection counts over matrices resembles the distribution of the LVF across matrices in Fig. 5. This is significant as injection experiments and LVF analysis thus validate each other. Slight differences can be attributed to the fact the LVF is based on failure-free execution while Fig. 6 is based on repeated executions for some corrections for certain detected errors (e.g., in the input matrices).

Observation 3: End-to-end resilience corrected all SDCs, i.e., 3 to 4 times as many as single-kernel conventional techniques.

Figure 7 depicts the corresponding results for task-based end-to-end resilience. We observe a similar distribution across matrices to Fig. 6, yet the number of faults lower since the task-based approach requires less time to execute. Consequently, fewer faults are injected at the same MTTF rate. Task-based injection counts that were corrected also loosely resemble the LVF in Fig. 5 for the same reasons as before, only that E is now indicated to be more prone to faults than C due to observed error corrections.

Observation 4: The LVF (without error injection) indicates the relative vulnerability of data structures.

Figure 8 depicts the average completion times after fault injection. All faults that resulted in an error were detected and corrected by end-to-end resilience. The pragma-based approach (left half) resulted in 8%–15% overhead for a fault rate from 45 to 25 s. Notice that such a high fault rate results in one or more faults per execution, some of which result in detectable errors that are subsequently corrected at the expense of executing recovery code. Again, such high SDC rates are not realistic, but they allow us to compare the *relative* overhead between pragma- and task-based. For the task-based case, overhead ranged from 8%–14%, nearly the same as pragma-based. The absolute time (y-axis) indicates that task-based is more efficient since tiling results in higher data reuse in caches on one hand and due to less overhead for corrections limited to a single tile on the other hand.

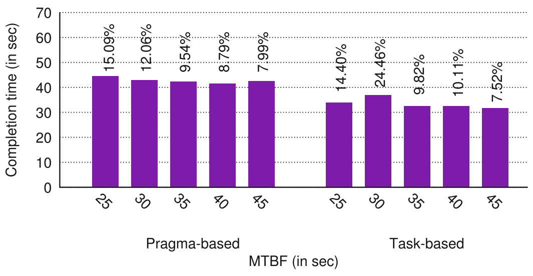

Fig. 8. Completion time with faults

Observation 5: Overall, pragma- and task-based resilience result in comparable overheads for matmult.

6.2 TF-IDF

We further assessed the resilience capabilities for an MPI-based benchmark. We ported a term frequency/inverse document frequency (TF-IDF) benchmark for document clustering based on prior work [38]. TF-IDF is a classification technique designed to distinguish important terms in a large collection of text documents, which is the basis for page ranking with applications in data mining and search engines. The classification is broken into two steps. (1) TF calculates the frequency of a term on a per document basis. (2) DF counts the number of occurrences of a given term (document frequency). The final result is $tfidf = TF \times log\frac{N}{DF}$. Note that TF is a local computation while DF is global across all documents. As a result, the DFs need to be aggregated.

Figure 9 depicts the steps in the TF-IDF benchmark. At first, the names of files are loaded. Then the term frequency (TF) method is called with `filenames` as input and `tfs` as output. Next, the document frequency (DF) is called with `tfs` as input and `dfs` as output. Finally, the *tfidf* value is computed for every term with a `TFIDF` call with `tfs` and `dfs` as input parameters. Figure 9 also depicts single kernel protection areas (arrows on left) and the vulnerability windows (live ranges) of variables protected by end-to-end resilience (arrows on right). The DF method contains MPI communication for the aggregation of document frequencies across all MPI ranks.

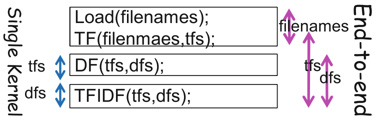

Fig. 9. Vulnerability windows in TF-IDF

Table 1. Compiler-derived resilience info

Region	Variable Name	Check method	Recover method
0	fn	Checker(fn)	Load(fn)
1	tfs	–	–
2	tfs	Checker(tfs)	Recover(tfs), Region(0)
	dfs	Checker(dfs)	Region(1)

Check and Recover Methods: TF-IDF does not have any check provided by the algorithm. Thus, we compute a checksum over the data. To demonstrate the capabilities of end-to-end resilience, we use a combination of redundancy and CR in this case study. CR provides a restore function, which we use as a recovery method.

Pragma Expansion: End-to-end resilience for TF-IDF can be provided by augmenting the code with three pragmas over as many regions (see Fig. 10). The first region is executed under redundancy with the default of dual redundancy to check if the filename is correct and, as recovery, reload the filename if this check fails. (Triple redundancy is optional and may allow the filename to be recovered, but this overhead is not justified just for protecting a filename.) The second region is protected with CR. The data of `tfs` is live across all three regions, while `dfs` is live across the last two pragma regions (`Continue` keyword).

```
vector<string> filenames;// input
vector<Dictionary> tfs;
// output of Region 1
map <string,int> dfs;
// output of Region 2

Load(filenames);
#pragma protect Redundancy Check(Checker(filenames))\
                           Recover(Load(filenames)) \
                           Continue
  TF(filenames, tfs);
#pragma protect CR Check(Checker(tfs)) Comm Continue
  DF(tfs, dfs); //contains MPI calls
#pragma protect Check(Checker(tfs), Checker(dfs))
  TFIDF(tfs,dfs);
```

Fig. 10. TF-IDF with protect pragma

Inside the DF method, MPI communication is used and, consequently, the `Comm` keyword is added to the second pragma. Table 1 depicts the regions, the input variable(s) to each region and the check and recover method per variable. Note that `tfs` is still live in region 2. Thus, no check should be carried out on `tfs` in region 1. Thus, region 1 does not have check/recover methods. The chaining of regions is also shown in Table 1. In region 2, `tfs` can be recovered by recomputing region 0. Similarly, `dfs` can be calculated from region 1.

We perform the code transformation in two steps. At first, the function calls for the CR or Redundancy schemes are added to the source code, followed by expansion of the pragma regions which provide end-to-end resilience. (final code omitted due to space).

Experimental Results of TF-IDF: We used 750 text books with a total size of 500 MB for the TF-IDF benchmark with 4 MPI ranks. We performed the evaluation with 4 input sizes: 125 MB, 250 MB, 375 MB, and 500 MB, which were protected by checksums.

Figure 11 depicts the time for conventional per-kernel resilience of TF-IDF and compares that to our end-to-end resilience. Execution times are averaged over 30 runs with small standard deviations (0.01–0.22). The overheads are almost the same, fluctuations of higher/lower execution by 0.25% or less are insignificant for input sizes of 125 MB to 16.2% for 500 MB. *This confirms observation 1.*

Figure 13 depicts the LVF metric on a logarithmic scale (y-axis) for the three kernels filenames (filen), tfs, and dfs and an input of 500 MB. The tfs data has the highest vulnerability. This reflects a combination of data size (tfs is larger than filenames/dfs) and live range of tfs during program execution (see Fig. 9). The other two kernels operate on smaller data and live ranges, and while this data still critical for resilience (e.g., names of files that will be opened), they add little overhead and are less prone to corruption (lower LVF). We observe again significantly increased protection ranges with end-to-end resilience at virtually unchanged overheads (0.01% to 2.71%). *This confirms observation 2.*

Similar to the fault injection code for matrix multiplication, we inject faults uniformly across the 3 data structures with fault rates from 25 to 45 s for TF-IDF. Figure 14 depicts the faults normalized against the respective data structure sizes.

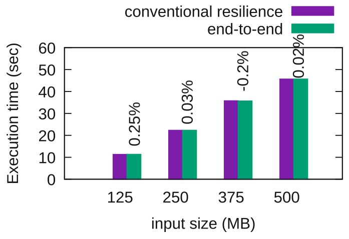

Fig. 11. Failure-free exec. time of TF-IDF

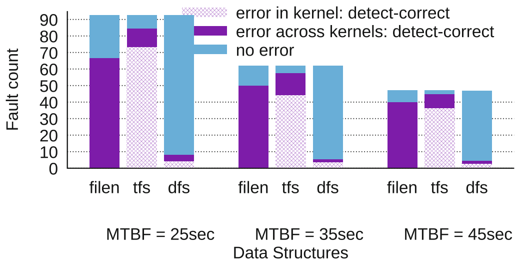

Fig. 12. Fault injection, various fault rates, 100 actual runs

The filenames data structure is small compared to the tfs and dfs structures, i.e., fewer faults are injected into filenames even though it has a larger life range than dfs. Similarly, tfs has the most injections as it is the largest data structure and is also live for the longest period of time. Finally, dfs is live for the shortest period of time, but because of its larger data footprint we see several injections into it. The shape of the fault distribution of Fig. 14 for actual injections closely resembles that of the modeling via the LVF metric in Fig. 13. *This confirms observation 4.*

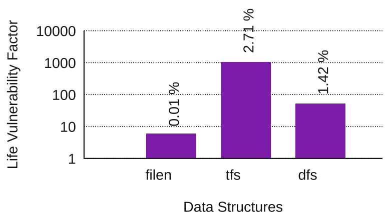

Fig. 13. Live Vulnerability Factor (bars) and % increase (above bars) for TF-IDF

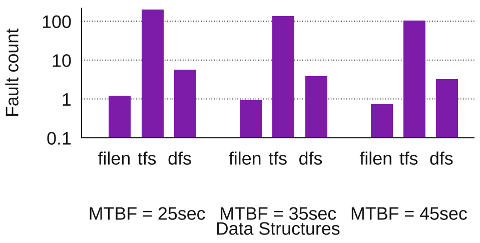

Fig. 14. Fault injection normalized to respective data structure sizes

Results indicating different fault handling classes are presented in Fig. 12. As with the matrix multiplication example, only the faults in the lower-most shaded regions of the tfs and dfs data structures can be corrected by conventional resilience methods while end-to-end resilience manages to detect and correct *all* errors, even those crossing scope/kernel boundaries. Furthermore, tfs was benefiting the most from end-to-end resilience while conventional resilience in a single kernel left many SDCs in tfs and some in dfs undetected (reflecting the vulnerability per data structure expressed by the LVF in Fig. 13). *This confirms observation 3.*

Discussion: We also experimented with a XOR hash to protect the data structures of TF-IDF. To produce a plain text as input for XOR, key/value strings of the tfs data structure were concatenated per file before they could be hashed. Due to string concatenation, this resulted in an additional 10% performance overhead for a total increase in LVF by 13% compared to no protection. This increase in LVF is clearly inferior to the simple checksums with 2.71% LVF overhead (Fig. 13), which underlines the importance of designing resilience mechanisms that require *small* metadata and perform checks with *little* performance overhead. Otherwise, resilience mechanisms might actually *increase* the chance of SDCs (due to a larger data footprint vulnerable for a longer time), i.e., a 100% increase in LVF doubles the chance of SDCs (even though they might be caught and fixed with end-to-end resilience).

Observation 6: The LVF indicates (without error injection) that The change in LVF (in %) reveals if protection was effective or counter-productive.

6.3 NAS Parallel Benchmarks

We apply our resilience pragma to the NAS parallel benchmark (NPB) suite to calculate the overhead of our scheme. We use the C port of the OpenMP version of the NPB suite. We annotate each iteration of the benchmarks with a pragma that protects the input data structures. Notice that only four of the NPB benchmarks of this C-based OpenMP version compile correctly for input class C, i.e., we cannot report results for the remaining ones, because data structures that remain unmodified between iteration boundaries are too small for end-to-end resilience to be effective. The input data structures are verified with a simple scheme that calculates a checksum over the entire data structure.

The data structure is checked for correctness inside the resilience region specified by the pragma and control is allowed to move to the next iteration only when the input data structures are verified to be correct. Via checkpointing, one can revert the data structures to a previously saved state in case a soft error is detected in the input data structures.

Figure 15 shows the execution time (in seconds) of the different benchmarks with and without the resilience pragma. The results are averaged over 10 runs of the benchmark and run for up to 30 min. We observe that the incurred overhead in case of a fault-free execution is between 0.63%–2.61%. The standard deviation for the executions ranges from 0.983 for FT to 9.957 for SP (too small to show in the plots).

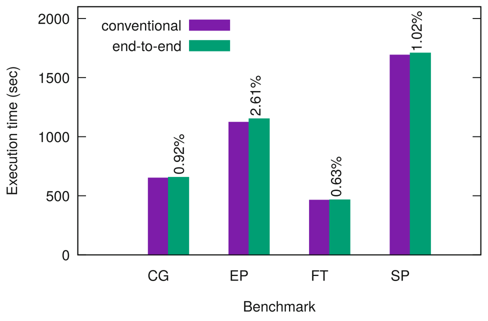

Fig. 15. Time of NPB codes (input class C) with/without pragma

Observation 7: Correctness of program-protected data structures can be guaranteed with a small penalty on performance of less than 3%.

7 Related Work

LVF differs from other metrics that assess resilience. The Failures in Time (FIT) rate is defined as a failure rate of 1 per billion hours. FIT is inverse proportional to MTBF (Mean Time Between Failures). The Architectural Vulnerability Factor (AVF) [35] is the probability that a fault (in microprocessor architecture) leads to a failure (in the program), defined over the fraction of time that data is vulnerable. The Program Vulnerability Factor (PVF) [34] allows insight into the vulnerability of a software resource with respect to hardware faults in a micro-architecture independent way by providing a comparison among programs with relative reliability. The Data Vulnerability Factor (DVF) [37] considers data spaces and the fraction of time that a fault in data will result in a failure. DVF takes into account the number of accesses as a contributor to fault occurrence. Past work has taken value live ranges into account to design a fault injection framework and measure CPU vs. GPU vulnerabilities in terms of PVF in Hauberk [36] and to protect critical data for GPU kernels [23]. Value live ranges encapsulate the live time of variables promoted to *registers* for short program segments while our variable live range captures the live time of compound structures/arrays over the entire program (from first define to last use) irrespective of register promotion. This is necessary as a singular structure/array element cannot be checked in isolation as required by end-to-end resilience (see next section), it can only be checked in conjunction with a subset of structure/arrays elements. Our LVF metric captures this difference and is thus different from AVF, PVF, and DVF. Furthermore, LVF takes into account time $\times$ space, which covers the effect of soft errors. Our metric is agnostic to architectural aspects of a processor (covered by AVF) and their impacts on programs (see PVF). It is also agnostic of the number of references (unlike DVF) as it considers both (a) written, incorrect results and (b) SDCs that may occur, even in the absence of write instructions (which other work does not). Simon et al. [32] use a Poisson distribution over a task's lifetime to determine the probability of task failures and derive from it the need for task-based replication. Unlike our work, they do not address the issue of data vulnerability when applications mix *multiple* resilience techniques. Diniz et al. [11] propose a resilience pragma to protect a *single* kernel. In contrast, our work contributes protection for end-to-end resilience *across* kernels.

8 Conclusion

We proposed an annotation-based approach for building highly modular and resilient applications such that resilience concerns are separated from algorithms. Our approach requires a minimal effort by application programmers and is highly portable. We introduced and investigated the significance of the live vulnerability factor, which takes into account the live range of a data structure and its storage space to provide insight into the likelihood of failures. We introduced an effective set of building blocks for detection and correction of soft faults through *Check* and *Recover* methods for arbitrary data structures. We provided two approaches, pragma- and task-based, to implement end-to-end resilience. We showed the effectiveness of end-to-end resilience for two variants of sequentially composed matrix multiplications and TF-IDF under failure-free execution and fault scenarios in detail. We further assessed the overhead of our method for NPB codes operating at input size C. End-to-end resilience incurred less than 3% overhead on average compared to conventional single-kernel resilience and increased protection against bit flips by a factor of three to four. The LVF metric helped in guiding which data structures to protect and assessing if protection meta-data and checking algorithms were effective (or counter-productive) in providing resilience.

References

1. Anderson, J.H., Calandrino, J.M.: Parallel task scheduling on multicore platforms. SIGBED Rev. **3**(1), 1–6 (2006)
2. Biswas, S., Supinski, B.R.D., Schulz, M., Franklin, D., Sherwood, T., Chong, F.T.: Exploiting data similarity to reduce memory footprints. In: IPDPS, pp. 152–163 (2011)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: PPoPP, pp. 207–216 (1995)
4. Böhm, S., Engelmann, C.: File I/O for MPI applications in redundant execution scenarios. In: Parallel, Distributed, and Network-Based Processing, February 2012
5. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Dongarra, J.: PaRSEC: exploiting heterogeneity to enhance scalability. Comput. Sci. Eng. **15**(6), 36–45 (2013)
6. Cao, C., Herault, T., Bosilca, G., Dongarra, J.: Design for a soft error resilient dynamic task-based runtime. In: IPDPS, pp. 765–774, May 2015
7. Chen, S., et al.: Scheduling threads for constructive cache sharing on CMPs. In: SPAA, pp. 105–115 (2007)
8. Chen, Z., Wu, P.: Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. IEEE TPDS **99**(PrePrints), 1 (2014)
9. Chung, J., et al.: Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In: Supercomputing, pp. 58:1–58:11 (2012)
10. Dave, C., Bae, H., Min, S.J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: a source-to-source compiler infrastructure for multicores. Computer **42**(12), 36–42 (2009)
11. Diniz, P.C., Liao, C., Quinlan, D.J., Lucas, R.F.: Pragma-controlled source-to-source code transformations for robust application execution. In: Desprez, F., et al. (eds.) Euro-Par 2016. LNCS, vol. 10104, pp. 660–670. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58943-5_53
12. Du, P., Bouteiller, A., Bosilca, G., Herault, T., Dongarra, J.: Algorithm-based fault tolerance for dense matrix factorizations. In: PPoPP, pp. 225–234 (2012)
13. Duell, J.: The design and implementation of Berkeley Labs Linux Checkpoint/Restart. Technical report, LBNL (2003)
14. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parall. Process. Lett. **21**(2), 173–193 (2011)
15. Elliott, J., Hoemmen, M., Mueller, F.: Evaluating the impact of SDC on the GMRES iterative solver. In: IPDPS, pp. 1193–1202 (2014)

16. Elliott, J., Kharbas, K., Fiala, D., Mueller, F., Ferreira, K., Engelmann, C.: Combining partial redundancy and checkpointing for HPC. In: ICDCS, 18–21 June 2012
17. Fiala, D., Mueller, F., Engelmann, C., Ferreira, K., Brightwell, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: Supercomputing (2012)
18. Geist, A.: How to kill a supercomputer: dirty power, cosmic rays, and bad solder. In: IEEE Spectrum, February 2016
19. Huang, K.H., Abraham, J.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Comput. **C–33**(6), 518–528 (1984)
20. Islam, T.Z., Mohror, K., Bagchi, S., Moody, A., de Supinski, B.R., Eigenmann, R.: MCREngine: a scalable checkpointing system using data-aware aggregation and compression. In: Supercomputing, pp. 17:1–17:11 (2012)
21. Kale, L.V., Krishnan, S.: Charm++: a portable concurrent object oriented system based on c++. In: OOPSLA, pp. 91–108 (1993)
22. Kiczales, G., et al.: Aspect-oriented programming. In: ECOOP, pp. 220–242 (1997)
23. Li, S., Sridharan, V., Gurumurthi, S., Yalamanchili, S.: Software-based dynamic reliability management for GPU applications. In: Workshop in Silicon Errors in Logic System Effects (2015)
24. Martsinkevich, T., Subasi, O., Unsal, O., Cappello, F., Labarta, J.: Fault-tolerant protocol for hybrid task-parallel message-passing applications. In: Cluster Computing, pp. 563–570, September 2015
25. Min, S., Iancu, C., Yelick, K.: Hierarchical work stealing on manycore clusters. In: Partitioned Global Address Space Programming Models (2011)
26. Moody, A., Bronevetsky, G., Mohror, K., Supinski, B.R.D.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Supercomputing, pp. 1–11 (2010)
27. Panzer-Steindel, B.: Data integrity. Technical report, 1.3, CERN (2007)
28. Parr, T., Quong, R.: ANTLR: a predicated. Softw. Pract. Exp. **25**(7), 789–810 (1995)
29. Schroeder, B., Gibson, G.A.: A large-scale study of failures in high-performance computing systems. In: DSN, pp. 249–258 (2006)
30. Schroeder, B., Pinheiro, E., Weber, W.D.: Dram errors in the wild: a large-scale field study. SIGMETRICS Perform. Eval. Rev. **37**(1), 193–204 (2009)
31. Shantharam, M., Srinivasmurthy, S., Raghavan, P.: Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In: Supercomputing, pp. 69–78 (2012)
32. Simon, T.A., Dorband, J.: Improving application resilience through probabilistic task replication. In: Workshop on Algorithmic and Application Error Resilience, June 2013
33. Snir, M., et al.: Addressing failures in exascale computing. Int. J. High Perform. Comput. (2013)
34. Sridharan, V., Kaeli, D.: Eliminating microarchitectural dependency from Architectural Vulnerability. In: HPCA, pp. 117–128, February 2009
35. Sridharan, V., et al.: Memory errors in modern systems: the good, the bad, and the ugly. In: ASPLOS, pp. 297–310 (2015)
36. Yim, K.S., Pham, C., Saleheen, M., Kalbarczyk, Z., Iyer, R.: Hauberk: lightweight silent data corruption error detector for GPGPU. In: IPDPS, pp. 287–300 (2011)
37. Yu, L., Li, D., Mittal, S., Vetter, J.S.: Quantitatively modeling application resilience with the data vulnerability factor. In: Supercomputing, pp. 695–706 (2014)

38. Zhang, Y., Mueller, F., Cui, X., Potok, T.: Large-scale multi-dimensional document clustering on GPU clusters. In: IPDPS, pp. 1–10, April 2010
39. Zheng, Z., Chien, A.A., Teranishi, K.: Fault tolerance in an inner-outer solver: a GVR-enabled case study. In: Daydé, M., Marques, O., Nakajima, K. (eds.) VECPAR 2014. LNCS, vol. 8969, pp. 124–132. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17353-5_11