Robust Graph Traversal: Resiliency Techniques for Data Intensive Supercomputing

Saurabh Hukerikar, Pedro C. Diniz, Robert F. Lucas
Information Sciences Institute
University of Southern California
Marina del Rey, CA 90292
Email: {saurabh,pedro,rflucas}@isi.edu

***Abstract*—Emerging large-scale, data intensive applications that use the graph abstraction to represent problems in a broad spectrum of scientific and analytics applications have radically different features from floating point intensive scientific applications. These complex graph applications, besides having large working datasets, exhibit very low spatial and temporal locality which makes designing algorithmic fault tolerance for these quite challenging. They will run on future exascale-class High Performance Computing (HPC) systems that will contain massive number of components, and will be built from devices far less reliable than those used today. In this paper we propose software based approaches that increase robustness for these data intensive, graph-based applications by managing the resiliency in terms of the data flow progress and validation of pointer computations. Our experimental results show that such a simple approach incurs fairly low execution time overheads while allowing these computations to survive a large number of faults that would otherwise always result in the termination of the computation.**

I. Introduction

Data-intensive applications are an increasingly important part of High Performance Computing (HPC) workloads. Supercomputing systems have traditionally been designed for maximizing floating point performance. Applications in science and engineering span areas such as high energy and nuclear physics, chemistry and materials, nanotechnology, astrophysics, biology. These simulation applications and their solvers are typically floating point intensive and tend to be well-structured in terms of control flow and data flow patterns [1]. The LINPACK benchmark which, solves a dense system of linear equations, has been used as the yardstick of performance for HPC systems, and has been used in the TOP500 rankings [2] over the past two decades. Recently, large-scale, data driven analysis applications that solve discrete math problems are becoming increasingly important parts of the supercomputing workload. These applications tend to use a graph abstraction for information mining, social network analysis, medical informatics, computational geometry, genomics, pattern matching and discovery. The Graph500 [3] [4] list reaffirms the growing importance of this class of workloads.

The challenge of managing resilience is very important to HPC system designers, application programmers, and users of future exascale computing systems [5]. Exascale systems will likely have millions of compute nodes and billions of processing and memory components. Such numbers increase the likelihood of faults that cause system level failures. Recently 60% of system failures may be attributed to hardware failures, of which memory related faults contribute 40% [6]. As VLSI geometries scale in future process technologies, the reliability of systems is challenged by greater device density, lower supply voltage, as well as manufacturing variabilities. Technology scaling beyond 45nm feature sizes show a significant increase in the fraction of multi-cell upsets (MCU), where more than one device or bit are affected. The dramatic increase in memory capacity needed in future exascale systems to sustain the data intensive application workloads, and their massive working sets, suggest that its contribution to system failures will increase even further. HPC systems today typically employ ECC memories which offer single bit error correction and double bit error detection. These will unlikely to be scaled to handle correction of multi-bit cell upsets because of the overheads of memory access latency and chip area. There is also silent data corruption, which escapes hardware-based detection mechanisms and manifests itself through incorrect application results, or system failures whose causality is difficult to trace [7].

Graph problems tend to be very large and sparse, and optimization efforts emphasize efficiency in memory bandwidth and network bandwidth utilization [8]. The algorithms make non-contiguous, concurrent accesses to global data structures with low degrees of locality [9], and the data structures tend to be referenced indirectly and therefore are rich in pointer arithmetic. These features have important implications in managing the fault resilience of these applications. First, pointer-rich data structures are highly sensitive to memory failures [10]. Even single bit upsets in pointer variables may lead to invalid references, causing segmentation faults. Second, the non-contiguous arrangement of graph nodes implies that node structures that occur in contiguous memory locations may not necessarily be linked with edges. Therefore, any scheme that employs redundancy through naive replication of graph nodes will incur great overheads to the memory and network bandwidth, given that typical datasets today are of the order of several terabytes. Third, unlike array based data structures such as the matrices used widely in scientific applications that afford the opportunity to incorporate redundancy through parity or checksums on rows and columns, these graph-based data structures lack a similar, sufficiently general method for algorithmic detection and correction of faults. Each of these suggest that the challenge of managing fault tolerance for these graph modeled data intensive applications on supercomputing systems is a significant one.

In this paper, we propose software based resiliency techniques geared towards this class of applications. We propose

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

techniques where we can leverage the node and edge traversal data flow patterns of graph-based algorithms. When errors are detected, this allows for a temporal fault tolerance assesment of the significance of the node address where the error occured. We also explore the idea of selective duplication of pointer references through compiler based transformations. Additionally, we also propose compiler driven memory alignment of the graph node structures and their pointers as a way to validate pointer arithmetic operations and therefore detect silent data corruptions on these. These approaches provide resilient graph traversal since: (i) they allow the application to live with certain multi-bit errors that are detected but are uncorrectable by memory ECC schemes without crashing the application and, (ii) they provide robustness to silent data corruptions in pointer arithmetic and computation to which such applications are notoriously sensitive.

The rest of the paper is organized as follows: Section II describes the fault models and their significance to data intensive HPC applications. Section III describes the resiliency techniques while Section IV elaborates their respective implementation. Section V describes our experimental approach and section VI surveys related approaches in HPC resiliency.

II. Fault Characterization

As faults increasingly become the norm rather than the exception for high performance computing, there are two classes of faults that are anticipated to be particularly significant [5]. The first are multibit faults that are detectable by hardware based memory ECC schemes but cannot be corrected, since these schemes typically offer single error correction and double error detection. Since the hardware lacks correction capability, these result in non-maskable interrupts to the operating system which forces an application crash or the system to shutdown to prevent further corruption.

The second category of faults that is increasingly important is silent data corruption. These faults remain undetected by hardware-based schemes and ultimately manifest themselves as errors in the program state. The application may complete but with incorrect results, terminate with unexplained segmentation faults, or simply hang. Exacerbating the situation is the fact that in data intensive applications, a single fault may propagate. For example, a massive graph may include vertices and edges that span across machine nodes and the possibility of a single error causing a cascading pattern of corruption is high.

Besides the increasing vulnerability to multicell upsets of individual devices as process technology scales, the complexity of memory architectures with the inclusion of 3D die stacked DRAM and persistent memory is also increasing. This means that more advanced capabilities for hardware-based detection and correction of faults will not be without large overheads to chip area, access latencies and power. Furthermore, tracing the causality of silent corruptions and containing them before they lead to errors in the application state, will also be increasingly challenging.

The software-based resiliency techniques presented here aim to address each of these modes of faults, keeping in mind the access patterns of the massive datasets in these graph-based applications. The proposed techniques seek to provide low overhead and early detection of silent corruptions in pointer arithmetic, since these are computations to which these applications are especially vulnerable. The techniques also assist in creating application level knowledge on how to react to the anomalies in application state, both with silent faults and ECC errors. Additionally, programmer can guide application recovery or state amelioration through a callback mechanism. Although the techniques described in the following sections are for graph-based computations, they are equally applicable to more generic computations and other complex data structures.

III. Resiliency Techniques

The software-based approach presented here is based on compiler-generated resiliency that entails simple code extensions and transformations. This is supported by a runtime engine that controls application execution and termination.

A. Data Flow Progression: Tolerant Signatures

Large-scale, graph applications tend to have a wide data footprint that does not fit within the memory of single machine node. Furthermore, the edge-vertex traversals of most graph algorithms show distinctive patterns. For example, a search proceeds in a sequence of steps: each step iterates over a set of boundary vertices and enqueues the unvisited neighbors to be processed at the next depth. This kernel is a basic building block used in graph analysis and has similar data access pattern characteristics as other graph applications. The nodes tend to be accessed only once (or finite number of times) per traversal of the graph. Therefore the memory assigned to the nodes becomes irrelevant (or dead) from the application's perspective as the traversal progresses and nodes are dequeued after being visited. The nodes may continue to remain mapped in the memory hierarchy and experience a fatal ECC failure, or even a recoverable memory error. In either case, it is possible to ignore these errors in an increasing large set of graph nodes (and hence of memory footprint) as the computation progresses.

To be able to trace the fault tolerance of data with respect to the application progress, we introduce a signature word pattern to every graph node structure. Nodes that have been traversed are marked as being subsequently tolerant to an arbitrary number of errors in all the memory associated with that graph node. To accomplish this, a runtime system must be aware of the location of the signature word with respect to the storage layout of the graph nodes and can rely on word alignment knowledge to efficiently track which memory regions are incrementally being marked as tolerant.

This makes for a dynamic and application-specific approach to fault tolerance. In complex and long running graph analytics applications as the application progresses, an increasingly growing percentage of its dataset is traversed and becomes tolerant to errors. Figure 1 illustrates this notion with a sample graph traversal algorithm. This technique of tracking progress of the data structure traversal is especially useful with graphs that span petabytes in dataset size and streamed in to the memory, but whose graph vertices are traversed finite number of times. The unique feature of this approach is that it is independent of the starting vertex or any other structural feature of the graph such as order of individual vertices or how

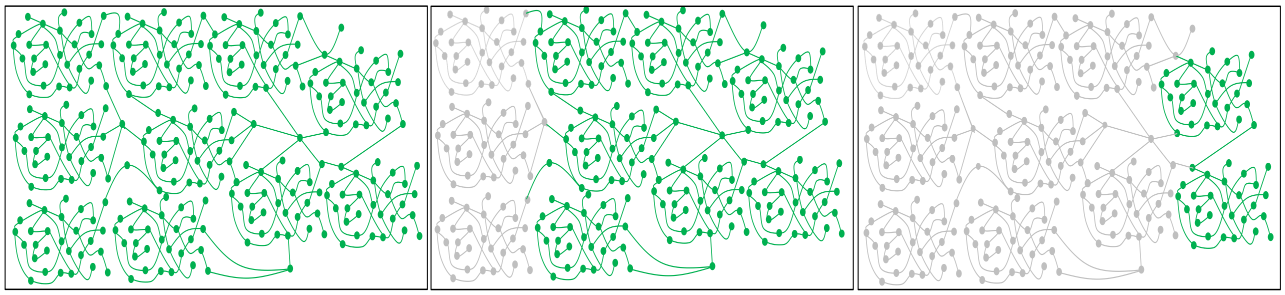


Fig. 1: Marking Nodes Tolerant: Shaded graph nodes are dynamically marked tolerant using a signature pattern as they become for the purposes of the current traversal ”dead”

sparse or dense the graph is, and yet allows reasoning about resiliency based on application level data flow rather than the physical address mapping in the memory hierarchy.

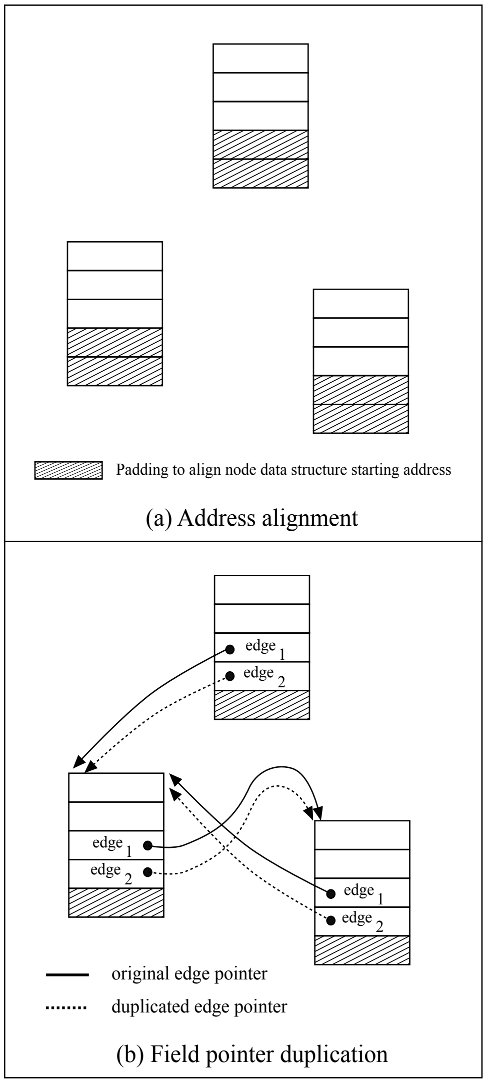
B. Memory Alignment

Data intensive applications built with complex data structures tend to be rich in pointer arithmetic and these computations are especially vulerable to silent corruptions, which may lead to segmentation fault induced crashes. Worse, faults can propagate to the end of the computation, leading to incorrect answers. For the detection of silent pointer corruption, we propose a scheme that enforces the alignment of the layout of the storage associated with nodes of a pointer-based data structure. We place a padding constraint on data structure objects and their constituent primitive data type elements laying out the nodes at specific address boundaries. This allows for the compiler to generate simple code that can detect silent faults in the pointer address bits during node traversal. This bit-level error detection and correction is accomplished using simple instruction sequences and thus with low performance overhead. This enforced alignment through padding is illustrated in figure 2(a).

C. Pointer Duplication

To complement the alignment approach, we also perform automatic compiler-based pointer field duplication. This approach is based on N-modular redundancy fault tolerance approach where the same computations are performed and then checked for inequalities among the N copies of the result. This approach incurs some storage and computation overhead for storing duplicated pointer fields and replication of the pointer-arithmetic. This is illustrated in figure 2(b) where the edge$_2$ pointer in all nodes are the shadow pointers. Through a simple compiler-driven code transformation, we duplicate all instructions that perform arithmetic on pointer addresses and compare the results to ensure that they match. The enforced address alignment may therefore also provide the means to choose among unequal result pointer values.

In the case of silent data corruption, detection of the fault is only part of the approach. One must also provide effective means for containment of the error and facilitating recovery. For managing recovery actions, we provide an application level handler function so that the application programmer has the ability to determine how to handle the error. The programmer



Padding to align node data structure starting address
(a) Address alignment
edge 1
edge 2
original edge pointer
duplicated edge pointer
(b) Field pointer duplication

Fig. 2: Pointer-based Resiliency Techniques: (a) Node Data Structure Alignment (b) Pointer-field Duplication.

may choose to gracefully terminate the application, or attempt to recover by attempting to restart the traversal from the previous level of the graph (repetition may sometimes account for transient faults), or may even choose to ignore the fault and isolate the graph node. This offers an effective containment strategy by ensuring that corruptions do not spread and cause further errors that may potentially lead to catastrophic application failure.

IV. Implementation

We now describe the implementation of the resiliency techniques described in Section III which are based on extensions to library calls and compiler based code transformations and supported by a runtime engine. The first technique which tracks the data flow progression during graph traversal specifically targets ECC memory errors that are detected, but may or may not be recoverable in hardware. The other two techniques (alignment and pointer field replication) target silent corruptions in pointer arithmetic. These later two techniques require additional code to proactively check for the alignment and consistency among duplicate copies of the pointers before a memory access is made. Also, the code needs to be extended so that the application is offered the opportunity to contain the fault and recover state whenever possible. We have automated these through compiler driven transformations using the ROSE source-to-source compiler infrastructure [11]. Some of the techniques described here are also applicable to more generic computations and data structures, besides pointer arithmetic, which may be driven by the programmer through preprocessor directives.

A. Library Extensions and Compiler Transformations

The feature that tracks the progression of graph node traversal is based on runtime memory allocation and is implemented as an extension to the library call malloc. The extended version of malloc, `rmalloc()` is implemented so that every block of memory allocated contains a special field that is appended to the block. The field is set with a signature byte sequence when the block is accessed. In the context of the graph traversal, as every node structure is accessed, the block of runtime memory associated with it is set with the signature pattern. The pattern is set such that it can reflect the count of the number of times the allocated block was accessed, so that the library knows when the memory block becomes irrelevant (and therefore tolerant to errors). Additionally, the extended version of malloc also inserts padding to align addresses and automatically duplicates the pointer types.

For the purposes of validation of pointer arithmetic against silent faults, we use the ROSE compiler, an open source infrastructure that enables source-to-source program transformation. It generates the intermediate representation (IR) as an uniform abstract syntax tree (AST). The source transformation pass uses a traversal function to discover the statements that perform the pointer arithmetic. The AST is then transformed to insert the statements that detect discrepancies in values of the instruction that perform pointer arithmetic.

For these transformations to be applicable to more general computations, we also provide `#pragma robust`. A naive compiler based replication of all instructions incurs too high an overhead. Using a directive, the programmer may explicitly annotate general computation statements that need to be duplicated and whose results also need to be compared to detect possible corruptions. An AST visitor function is used to gather the statements attached to the nodes encompassed by the `#pragma` statement.

The application code is also extended by registering a callback function for handling the error containment and recovery actions. The call back mechanism allows the programmer to decide the course of recovery: whether to terminate gracefully, or to restart algorithm from a previous level of the graph or even ignore a set of nodes from the traversal and resume at next unvisited neighbor.

```
typedef void (*ECC_callback_t)
      (void *add_location, void *arg);
```

```
void ecc_recover_init(ECC_callback_t cb, void *arg);
```

B. Runtime Support

For reasoning about the ECC memory errors, our approach requires runtime support to maintain a mapping of the physical addresses and address offsets. The runtime is an independent light-weight process that monitors the state of the graph application. Upon receiving notification of an uncorrectable memory error through an interrupt handler, the runtime system is signaled that an ECC unrecoverable error has occurred at a specified address. If it discovers that the unrecoverable error is mapped to a location that has been marked with the signature sequence and hence irrelevant to the application, it ignores the error and resumes the application, rather allowing the application or system to crash.

V. Experimental Evaluation

The kernel on which Graph500 results are ranked is Breadth-First Search, a representative graph traversal kernel on unweighted graphs used in our experiments. The evaluation platform is an Intel Xeon^{TM} 8-core 2.4 GHz machine.

A. Fault Coverage Analysis

The Graph500 benchmark consists of phases that include: the generation of an edge list, graph construction from edge list, followed by the traversal kernel. The traversal itself proceeds in a sequence of steps: each of which iterates over a set of boundary vertices and enqueues the respective unvisited neighbors to be processed at the next depth. As the vertices are visited and dequeued their respective fields is set with a signature pattern.

Our fault injection tool dynamically sweeps the active regions of the application's address space to evaluate which regions map to nodes that contain the signature byte sequence. These are the regions that can tolerate an arbitrary number of ECC errors and the scope of this tolerant memory tracks the progress of the application, as is evident from figure 3. In the first phase of graph construction, the percentage of tolerant regions remains at zero. In the traversal phase as the vertices are visited and then dequeued from the visitor list, their memory becomes irrelevant to the current state of the application. As

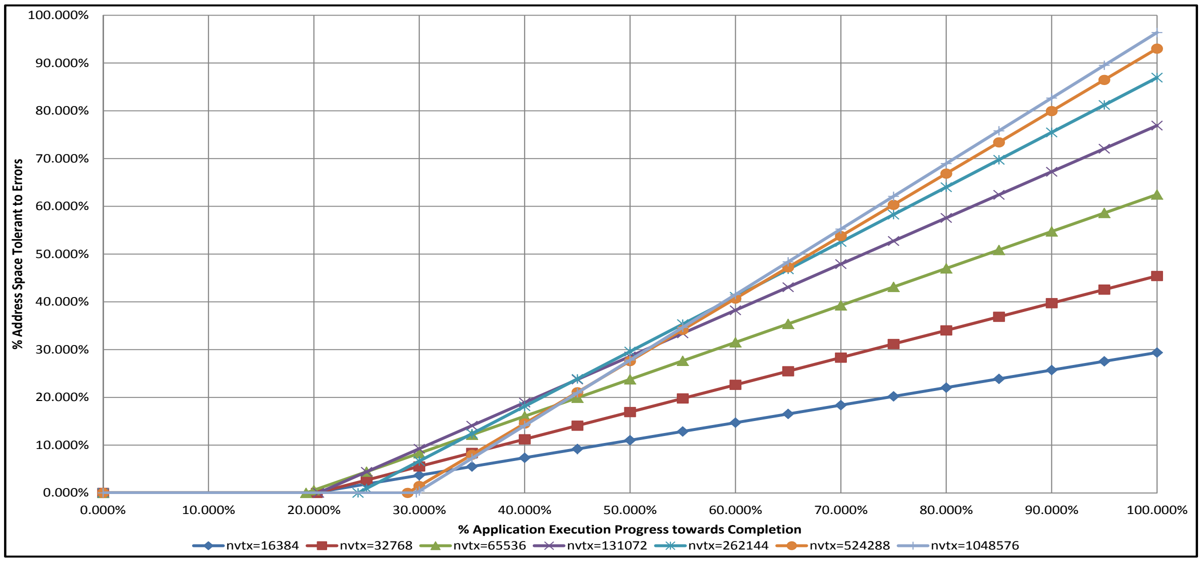


Fig. 3: Data Flow Progression: Fault Coverage Analysis

figure 3 reveals, the percentage of address space that becomes error tolerant progressively increases as application execution proceeds towards completion. The percentage never reaches 100% as the code sections, stack sections remain in the active memory for the lifetime of the application and these are not explicitly error tolerant. However, as the data set size increases, the fraction of tolerant memory reaches 98% for a graph with 2 million of vertices.

B. *Fault Injection Experiments*

1) Fault Injection for Data Flow Progression: To evaluate the effectiveness of the data flow progression technique for ECC errors, we use dynamic fault injection to inject faults into the active address space of the application while it is in execution. For these experiments, we inject one multi-bit ECC error per application run that raises an ECC failure interrupt and signals the runtime engine to handle application recovery or terminate. The runtime looks up the address mapping and checks for the signature sequence to determine if the error occurred at a location that the data flow has progressed beyond and if so, allows application to proceed. Otherwise the application terminates. For these experiments, we perform $10,000$ application runs per graph size, so that the Kronecker generator produces different edge lists whose layout in memory is different each time. Additionally, the fault injection is randomized in time and address space location across application runs. Fig 4(a) depicts the percentage of application runs that can tolerate an otherwise fatal ECC uncorrectable error and complete correctly. The figure also shows the correlation between survivability and graph size.

2) Fault Injection for Silent Data Corruption: In these experiments, the faults of interest are silent data corruptions including single-bit flips in memory undetected by ECC as well as those that occur when there multiple bit flips that ECC is unable to detect. Possible outcomes include: pointer segmentation fault leading to application crash; the fault manifests in the data variables such that the application completes but

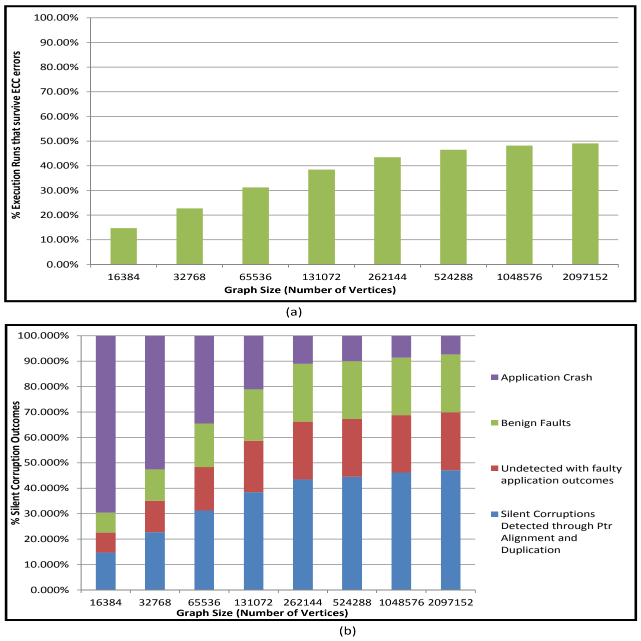


Fig. 4: Graph traversal resiliency techniques: (a) Survival rate using dynamic tolerant marking for ECC errors; (b) Fault injection outcomes using alignment and pointer duplication for Silent Data Corruptions.

produces incorrect results; benign faults that do not manifest in application outcome; or the application crashes because the application code was affected leading to illegal control flow. We are interested in the detection of these faults which can then initiate our callback mechanism. We evaluate whether our proposed approach based on memory alignment and pointer duplication can detect faults in pointer computations. Similar to the previous set of experiments, we perform 10000 application runs per graph size and our fault injection tool perturbs the victim bits and then observes the propagation of the faults. Figure 4(b) summarizes the results of these experiment runs.

C. Performance Evaluation

We measured the performance overhead of these techniques as the wall clock execution time difference between the computation with and without the proposed application code extensions. We found that the overhead in the time to solution is 1.2% or lesser for all the graph sizes we evaluated. Besides the overhead to execution time, the Graph500 benchmark also uses the metric of Trillion Edges traversed Per Second (TEPS) which reflects the bandwidth efficiency and the overhead to the TEPS rating for our techniques is 1.7% or lesser for all the graph sizes we evaluated. We attribute these low overheads to the reduced number of additional instructions per traversal step and to the fact that memory padding also adds a few bytes per graph vertex to enforce alignment.

VI. Related Work

Traditional supercomputing workloads including simulation applications and linear solvers, widely use Algorithm-based fault tolerance (ABFT) schemes to detect and correct errors. These approaches leverage the two-dimensional structure of matrices by incorporating redundancy at the row or column-level using checksums for common dense matrix operations [12] as well as for Cholesky factorization and LU-decomposition [13]. Other ABFT methods that are based on the algorithm such as iterative methods including Jacobi, Gauss Seidel, the conjugate gradient, and multigrid methods can leverage convergence properties of the algorithm for error detection and correction [14]. While such ABFT techniques tend to work well for numerically intensive structured computations, similar low overhead algorithmic error detection and correction for relatively complex unstructured graph modeled search and traversal algorithms tends to be much harder.

Much research has been conducted on the use of software based checkpoint and restart as a methodology to recover from failures whether at the node operating system level [15] or through checkpointing in message passing libraries such as FT-MPI or OpenMPI. A limitation of these approaches for data intensive applications is the large performance overhead incurred due to the volume of globally coordinated state needed to be captured and restored to/from persistent storage.

In commercial applications, data processing systems that process massive data sets of crawled documents, web request logs etc, tend to use the transaction model that performs atomic updates to a redundant database or storage system. The MapReduce library [16], used in various data mining and analytics applications, can handle failures of computational nodes by having a master node periodically ping the worker nodes and simply re-executing the entire task assigned to non-responsive failed node on a different worker node. Such approaches are complementary to ours, since they wait for node failure before reacting to any error in the application state and rely on redundant persistent storage for recovery.

VII. Conclusion

In this paper we described a set of software-based, compiler-driven resiliency techniques for data-intensive applications that manipulate graph data structures. The approach presented here leverages the data flow usage patterns and alignment properties of the graph node data structures, providing early detection opportunities for fault containment, amelioration of state or graceful termination before memory data corruption leads to catastrophic failures. We presented an experimental evaluation of the use of these techniques for large scale graph traversal computation. The results reveal that despite their simplicity, these techniques substantially increase the computation's survivability in the presence of undetected and uncorrectable memory errors and incur very low overheads to the application performance.

Acknowledgment

This research has been supported by the US Department of Energy (DoE) under the SciDAC-3 SUPER Research Institute (Contract Number DE-SC0006844) and Sandia National Laboratories (Award Number 1315083).

References

[1] R. Murphy and P. Kogge, "On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications," *IEEE Trans. Comput.*, vol. 56, no. 7, pp. 937–945, Jul. 2007.

[2] Top500 Supercomputer Sites. [Online]. Available: http://www.top500.org/

[3] R. Murphy, K. Wheeler, B. Barrett, and J. Ang, "Introducing the Graph 500," *Cray User's Group*, May 2010.

[4] Graph500. [Online]. Available: http://www.graph500.org/

[5] A. Geist, "What is the monster in the closet?" *Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking*, August 2011.

[6] S. Michalak, K. Harris, N. Hengartner, B. Takala, and S. Wender, "Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer," *IEEE Trans. on Device and Materials Reliability*, vol. 5, no. 3, pp. 329–335, Sep. 2005.

[7] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem: an architectural perspective," in *Proc. of the 11th Intl. Symp. on High-Performance Computer Architecture (HPCA)*, 2005, pp. 243–247.

[8] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 14:1–14:11.

[9] D. Bader, G. Cong, and J. Feo, "On the architectural requirements for efficient execution of graph algorithms," in *Proc. of the 2005 Intl. Conf. on Parallel Processing (ICPP)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 547–556.

[10] Y. Aumann and M. Bender, "Fault Tolerant Data Structures," in *Proc. of the 37th Annual Symp. on Foundations of Computer Science (FOCS)*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 580–589.

[11] ROSE Compiler Infrastructure. [Online]. Available: http://www.rosecompiler.org/

[12] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[13] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: a fault tolerant implementation without checkpointing," in *Proc. of the Intl. Conf. on Supercomputing*, 2011, pp. 162–171.

[14] A. Mishra and P. Banerjee, "An algorithm-based error detection scheme for the multigrid method," *IEEE Trans. on Computers*, vol. 52, no. 9, pp. 1089–1099, 2003.

[15] J. Duell, P. Hargrove, and E. Roman, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Berkeley Lab, Tech. Rep., Dec. 2002.

[16] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.