An Evaluation of Lazy Fault Detection based on Adaptive Redundant Multithreading

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

†Sandia National Laboratories
Livermore, CA, USA
Email: knteran@sandia.gov

***Abstract*—The challenge of resilience for High Performance Computing applications is significant for future extreme scale systems. These systems will experience unprecedented rates of faults and errors as they will be constructed from massive numbers of components that are inherently less reliable than those available today. While the use of redundant computing can provide detection and possible correction of errors, its system-wide use in future extreme-scale HPC systems will incur considerable overheads to application performance.**

In this paper, we present a framework that provides application level fault detection based on redundant multithreading. In previous work, we demonstrated an adaptive approach based on a language level directive. The computation contained in the programmer directive is executed by duplicate threads. In concert with a runtime system, the redundant multithreading is enabled opportunistically to provide fault detection at more reasonable overheads to application performance. The lazy fault detection approach presented in this work seeks to further optimize the use of redundancy by prioritizing the application's primary computation over the fault detection. Our approach relaxes the requirement that the redundant threads synchronize and compare results immediately. We show that lazy error detection is feasible and yields lower time to solution over adaptive RMT for a range of scientific computational kernels. We also explore a thread-to-core assignment strategy that seeks to reduce the interference between the redundant threads.

I. Introduction

Reliability of computations in High Performance Computing (HPC) applications is one the significant challenges for future exascale-class supercomputing systems. Recent trends suggest that future exascale HPC systems will be built from hundreds of millions of components organized in complex hierarchies to satiate the demand for faster and more accurate scientific computations [1]. However, with the growing number of components, the overall reliability of the system decreases proportionally. Furthermore, with process technology scaling, these systems will be constructed from VLSI devices which are less reliable than those used today. The consequent dramatic increase in fault and error rates threatens the validity of scientific simulations as well as limits the scalability of these applications. The impact on long running scientific applications is that they will either not finish, or worse, may complete, but produce incorrect results.

HPC applications today are neither fault aware nor fault tolerant, and current HPC programming paradigms are not explicitly designed to detect and contain errors to limit their propagation. The most widely used techniques for managing application resilience are based on Checkpoint and Rollback (C/R) which activate only upon failure of a process. The application state is recovered by killing all the remaining processes and restarting the application from the latest global checkpoint. Various studies have suggested that global C/R is not a viable solution for future massive scale HPC systems [2].

Algorithmic techniques offer the ability to detect and correct bit flip errors in part of the application state by encoding the data structures and adapting the algorithms to operate on the encoded data in the context of various linear algebra kernels [3] [4]. Redundant computation entails executing identical copies of the program code and enables detection (in the case of Dual Modular Redundancy (DMR)) or correction (in the case of Triple Modular Redundancy (TMR)) of errors by majority voting on the results produced. Implementations may use multiple hardware thread contexts, or compiler generated threads/processes or even be supported by Message Passing Interface (MPI) libraries [5].

There is a significant performance penalty associated with fault detection through redundant execution due to the duplicated computation which would limit its system-wide use in the context of long running scientific applications running on future exascale-class systems. However, with the rising frequency of faults and errors, application level detection and correction of errors is a much needed capability. In an effort to provide some balance between the two, we proposed Adaptive Redundant Multithreading (aRMT) in previous work [6] [7]. The approach permitted the application programmer to tailor the application's use of redundancy and therefore the extent of fault coverage. Additionally, redundant computation is enabled/disabled by a runtime system based on continuously observing and assessing the fault tolerance state of the system.

In this paper, we develop a *lazy* fault detection mechanism based on the insight that error detection may be off the critical path of the application's primary computation. The goal is to prioritize the application's primary computation over fault

978-1-4799-6233-4/14/$31.00 ©2014 IEEE

detection and therefore mitigate the impact of the redundant computation on the application's time to solution. Like the adaptive RMT, our compiler generates duplicate threads for the programmer defined regions of the application code. However, in this *lazy* scheme, we relax the requirement that the redundant threads synchronize and compare results before the application is allowed to progress. We also explore a strategy for assignment of the redundant threads to processor cores such that the redundant computation and the value comparison threads are assigned to a dedicated processor.

The rest of this paper is organized as follows: Section II describes the basic concepts of Adaptive Redundant Multithreading (aRMT) while Section III explains our lazy fault detection approach and explores a strategy for thread-to-core assignment to support the lazy detection. Section IV describes our experimental evaluation and performance results. Section V surveys related RMT based fault detection and correction approaches.

II. Adaptive Redundant Multithreading

The basis of our approach is Redundant Multithreading (RMT) which has been used as a common form of fault detection [8]. RMT entails either complete or part duplication of program instructions using independent threads and comparing the respective outputs to detect the presence of errors. The extent of the program code that is executed redundantly may be captured in a logical construct called the *Sphere of Replication* (SoR) [9]. The SoR represents a logical boundary that includes an unit of computation and any fault that occurs within the sphere of replication propagates to its boundary. Faults are detected by comparing specific outputs produced by the redundant multithreaded execution of the SoR.

The key considerations in the design of such spheres of replication entail identification of regions in the HPC application code whose redundant execution outcome will indicate the presence of errors in the computation. Also critical is identifying the input variables that need to be included in the sphere of replication and the output variables from the redundant threads that need to be compared to ensure that the fault coverage is not compromised and that all errors are identified.

Our previous work on adaptive redundant multithreading (presented in [6] and [7]) leverages the insight of application programmers who often tend to be well-positioned to understand the fault coverage requirements of their application codes. We presented a programming directive that is based on a preprocessor pragma directive that enables programmers to define spheres of replication in their application code. The syntax for the directive is:

```
#pragma robust private(variable list...)
               shared(variable list...)
               compare (variable list...)
{
  /*
   code
   ...
  */
}
```

The code that is textually enclosed between the beginning and the end of the code block following the directive may be executed by redundant threads. The scoping clauses `private` and `shared` allow binding the scope of the variables to the duplicate threads, whereas the `compare` clause lists variables whose values are compared to detect errors. Further details on the compiler outlining of these structured code blocks and the runtime actions to activate and disable the redundant execution are detailed in [6] and [7].

III. Lazy Fault Detection

In this section, we describe the mechanics of the lazy fault detection and the impact on application performance as well as the specifics of the implementation. Next, we explore a thread-to-core assignment strategy and finally, discuss the implications of the lazy approach for error recovery.

A. Infrastructure for Fault Detection

The redundant multithreading based on the programmer directive provides opportunistic fault detection with the support of a runtime system. The structured code block is outlined by the compiler but is executed by a single thread until the runtime system intervenes. The runtime system gathers information on the fault tolerance state of the system through events such as corrected ECC notifications, PCI bus parity errors, DRAM scrubbing notifications etc. Based on the frequency of such events, the runtime system makes inferences about the vulnerability of the system resources and signals the application to dynamically enable/disable the redundant multithreaded execution of the `#pragma` delineated code blocks.

This opportunistic approach has two key features: first, it permits the application developer to tailor the extent of the coverage; and second, the redundant execution is enabled only when the rate of anomalous events rises to the point where application level fault detection is deemed necessary. The concept is illustrated in Figure 1 (a) for a typical iterative solver code where the occurrence of an event causes the runtime to enable RMT. Each trapezoidal structure in Figure 1 (a) represents a fork of duplicate threads that execute the code block and synchronize and compare the output variables. Following an anomalous event, the iterations are executed in RMT and the partial solution is compared before proceeding to the next iteration.

The amount of computation that may be enclosed within the programmer defined sphere of replication is highly application dependent. For applications, where long phases are enclosed in the pragma block, or in the case of compute bound applications, the imposition that output values be compared after both the duplicate threads complete incurs nontrivial overhead to the overall performance. To address this limitation, we propose *lazy* fault detection where we relax the requirement that the redundant threads synchronize immediately after the execution of the code block. The *lazy* strategy is based on the insight that the detection of the error need not necessarily lie on the application's critical path of execution. This requires a buffer space where the duplicate threads can write their respective output values as they execute each block. A separate lightweight fault detection thread performs the value comparison. This approach is illustrated in Figure 1(b) for an iterative solver code, where the occurrence of an anomalous event initiates the RMT execution. However, in this

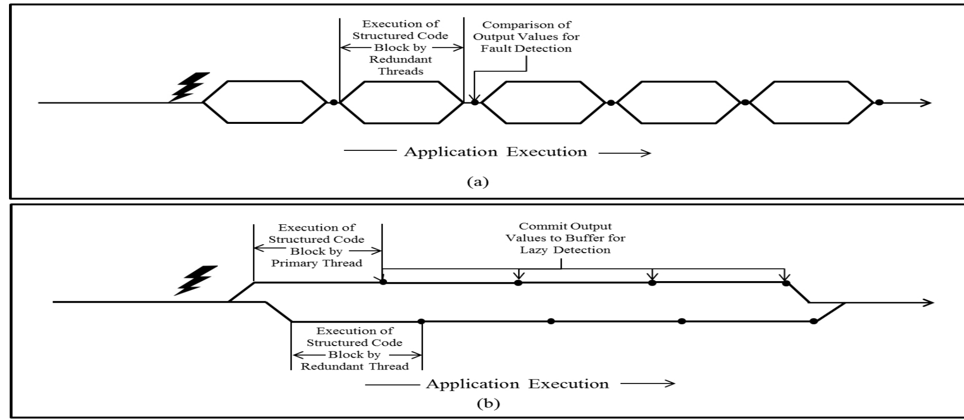

Fig. 1. Execution Timeline of RMT: (a) Adaptive Redundant Multithreading (b) aRMT with Lazy Fault Detection

case the duplicate threads can execute subsequent iterations independently, without needing to synchronize for output value comparison.

The key benefit of relaxing this requirement is that the threads can be differently prioritized while scheduling them. By elevating the scheduling priority of one the threads, called the *primary* thread, allows its subsequent iterations to be run at higher priority. The other thread, called the *redundant* thread, performs the duplicate computation for the same iterations but is scheduled to run at lower priority. The output value comparison is done by a dedicated thread which is also assigned a lower priority. The multi-level priority assignment to the *primary* threads helps mitigate part of the overhead caused by RMT.

B. Implementation

A source-to-source compiler pass identifies and outlines the `#pragma robust` structured code blocks. The outlining entails extracting the code block into a function. The compiler generates code to fork the duplicate threads to which the outlined function pointer is passed. The assignment of the thread priority is also handled by the compiler. Most common threading libraries allow statically assigning the scheduling priority of the threads via a parameter flag. Among the duplicate threads, the *primary* thread is assigned an elevated priority over the *redundant* thread. A detection thread is forked for the purpose of comparing the values that are committed to the result buffer and this thread is also statically assigned to be scheduled at low priority.

The compiler inserts code for the allocation of buffer space into which the duplicate threads can commit their respective outputs. Each buffer entry contains a pair of elements. The buffer is implemented as a cyclic FIFO and the size of each buffer entry is determined based on the data types of the variable list specified in the `compare` scoping clause. Also, associated with each individual element in the pair is a flag to set its validity. Also, each buffer entry has another flag that tracks whether the buffer entry i.e. the element pair is currently *active*.

We implemented a lock-free variant in which the code to compare the values was included within the outlined function body. The comparison was only performed when the thread was the last to update the buffer entry. This guaranteed that the comparison was performed only when both elements of the buffer entry were valid. This variant does not require a separate detection thread. In the implementation variant which tasked a separate detection thread with the value comparison, every circular buffer entry is protected by a lock. When the detection thread is scheduled, it performs value comparison for all *active* buffer entries where both the elements in entry pair are valid.

C. Core Grouping

Thread scheduling algorithms for traditional multiprocessors seek to evenly distribute the application workload over the available processors in order to maximize resource utilization. We propose creating an unbalanced scheduling based on core grouping that fits the lazy fault detection model. The implication of such an unbalanced thread-to-core mapping strategy is that it reduces the processor resources available to the primary computation. However, in our context, it also helps reduce the interference between the *primary* threads and *redundant* threads by grouping their mapping to processor cores. All the *redundant* threads as well as the detection thread are clumped together and assigned to a separate processor core, whereas the *primary* threads are balanced among the remaining processor cores on the SMP. The core grouping is performed statically and the compiler sets the CPU affinity of *redundant* and the detection threads to a specific processor core. The remaining processor cores in the system are grouped in a CPU set and the affinity of the *primary* threads is assigned to this CPU set. We observe that the effect of the unbalanced mapping is that the core which is assigned the redundant threads tends to be oversubscribed. However, since the lazy model promotes the execution of the redundant work off-critical path, the unbalanced core grouping aids in improving the application's overall performance.

D. Managing Error Recovery/Mitigation

The emphasis of this paper is on application level fault detection and in the current implementation any mismatch in output values causes the application to signal the runtime. The application is then gracefully terminated by the runtime. However, for other recovery mechanisms the lazy detection approach has implications since the the error may be detected

by the detection thread much after the execution of the primary structured code block.

For rollback recovery, the lazy detection can still initiate recovery much before catastrophic application failure. However, the present approach will need to be extended to ensure that the checkpoint state committed is consistent. In the case of preemptive fault avoidance strategies, such as fault-driven task migration, the lazy detection can be used to guide the task migration policy.

IV. Evaluation

In this section, we evaluate the performance of the lazy fault detection. We describe the opportunities exploited for RMT in popular scientific kernel codes and the experimental methodology. We then present the overall performance results with fault injection for these codes.

A. Experimental Setup

We evaluated our lazy detection framework on an Intel TMXeon 8-core 2.4 GHz compute node running the Linux operating system in the USC HPCC cluster. Our evaluation used five scientific computational kernels that form the core of many large-scale scientific applications. The use of the `pragma robust` for each code is summarized below. Our implementation uses the ROSE [10] source-to-source compiler infrastructure to identify and outline the `pragma` structured code blocks.

1) Double Precision Matrix Multiplication (DGEMM): For the double-precision matrix-matrix multiplication (DGEMM) kernel, we define the pragma block to include the inner dot product of the matrix multiplication *i.e.* the dot product computation resulting from the multiplication of a single row and single column of the operand matrices. When this code block is executed by multiple redundant threads, the reduced dot product is compared to detect the presence of errors in the computation.

2) Sparse Matrix Vector Multiplication (SpMV): In the case of the Sparse Matrix Vector Multiplication (SpMV), we enclose the body of the outer for loop i.e. the inner product into the `#pragma robust` structured code block, which is then outlined as a sphere of replication by the compiler. The reduced vector element for every row is the output value that is compared to detect the presence of errors when an iteration of the outer loop is executed by duplicate threads.

3) Conjugate Gradient (CG): The Conjugate Gradient (CG) method is an iterative algorithm that solves a system of linear equations and is implemented such that the initial solution is iteratively refined. The iterations provide a monotonically decreasing error norm and therefore an improving approximation to the exact solution. By enclosing each iteration in the block contained by the `#pragma robust` directive, the error value is compared when the iterations are executed by duplicate threads to detect the presence of errors in the solution.

4) Self-Stabilizing Conjugate Gradient (SSCG): The self-stabilizing approach to the conjugate gradient method [11] identifies a condition that must hold at each iteration for the solver to converge. Although other algorithmic detection techniques are possible, we include the solver code in the structured code block for adaptive redundant execution and the correctness of the computation is verified by comparing the error norm at the end of each iteration.

5) Multigrid Solver: In this hierarchical algorithm, the solution is accelerated by solving a coarse approximation of the original problem and interpolating it back to finer level and then refining that solution until it forms a sufficiently precise solution. We use a V-cycle depth of 8 and the `#pragma robust` directive is used to separately provide fault detection coverage to the restriction, relaxation and interpolation phases of the V-cycle.

The fault model is based on notifications for anomalous fault events such as corrected ECC and parity errors which are recoverable in hardware or by the system software. The notifications communicated through interrupts only serve to inform the runtime system. Accordingly, the fault events do not cause application or system failure, nor do they perturb any aspect of the application state or the computation. Our fault injection framework simulates such events by sending USR1 signals to the runtime system.

B. Performance Evaluation

We seek to understand the performance impact of the lazy fault detection on the application's time to solution complementary to the adaptive redundant multithreading. We evaluate explicit shared memory multithreaded implementations of the application codes described above. Since our framework is adaptive, the runtime determines whether and when to enable or disable the redundant execution. However once RMT is enabled, the duplicate threads run with lazy fault detection. For these experiment runs, we measure the time to solution of the application as the execution time until the primary computation threads converge. We then study the performance overheads for the thread-to-core assignment where the runtime groups all the duplicate threads and detection thread to a single processor core in an effort to reduce the interference for the rest of the cores in the SMP.

We experiment with five different fault event rates of 1, 2, 3, 4 and 5 events per execution respectively. The generation of the fault events is randomized, both in terms of the instant of generation of the initial event and the interval between subsequent events. Since the execution times for these computational kernels are very small, we conduct fault injection experiments that include thousands of application runs with randomized fault event generation. For each fault event rate, we perform 10000 application runs and compare the respective normalized average execution times.

Figure 2 presents an overview of the performance of the adaptive redundant multithreading with lazy detection for the five computational kernels. The baseline is a fault free execution run (represented by the 1.0x data points). Among the remaining data points, we can compare the average normalized application execution times (for the 10000 experiment runs) with the basic adaptive RMT and adaptive RMT with lazy detection. The figure makes the comparison for rates of 1, 2, 3, 4 and 5 random fault events per application run. For a rate of one fault per execution run, the adaptive RMT (without lazy detection) adds minimal overhead to the execution time

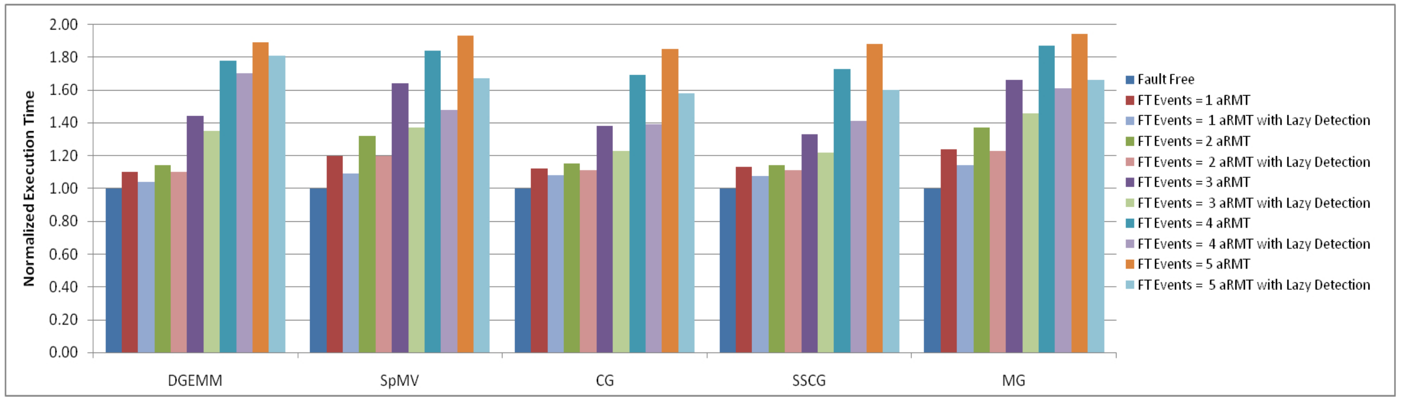

Fig. 2. Results: Performance Evaluation of aRMT with Lazy Fault Detection

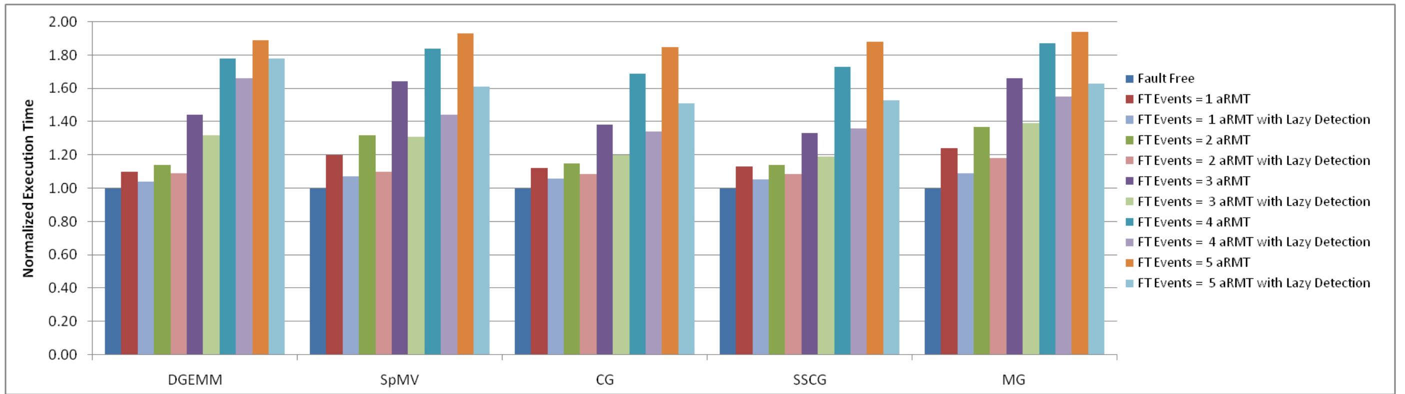

Fig. 3. Results: Performance Evaluation of Lazy Fault Detection with Core Grouping

since only a few iterations in each of the application codes are executed redundantly. For higher fault rates, the adaptive RMT is enabled for a longer part of the total execution time and accordingly demonstrates overheads ranging from 1.6x to 1.89x over the baseline. The adaptive RMT with lazy detection strategy provides further optimization in terms of time to solution over the basic adaptive RMT. The times to solution for DGEMM, SpMV, CG, SSCG, MG are 5.5%, 9.1%, 6.6%, 6.9% and 7.6% lower (for fault rate = 1 per execution). The benefit of the lazy detection is highlighted in the experiment runs with higher fault rates during which the RMT remains enabled for a larger part of the total computation. The execution times for the lazy adaptive RMT are 7.3%, 13.4%, 14.6%, 14.8% and 15.4% lower (for the fault rate = 5 per application execution) than the basic adaptive RMT.

Figure 3 shows the average normalized execution times when the lazy approach is supported by the core grouping strategy in which all the redundant threads as well as the fault detection thread are mapped to a dedicated core. The asymmetry aids in reducing the interference between the primary and the redundant threads yielding an improvement of 3 to 5% for fault rate = 1 per execution over the lazy adaptive RMT execution times. For a fault rate = 5 per execution, the execution times are a further 5.1%, 3.8%, 3.7%, 3.8% and 3.9% lower than the lazy adaptive RMT execution times (without core grouping). The overall benefit of the lazy fault detection together with the core grouping yields as much as 12.4%, 17.2%, 18.3%, 18.6% and 19.3% lower execution times respectively in comparison to the basic adaptive RMT.

V. Related Work

Hardware-based redundant multithreading was proposed to mitigate the costs of implementing complete hardware replication used by lock stepped processors such as the IBM G5 [12]. Early hardware-based RMT approaches proposed concurrently executing multiple explicit copies of the program code by sharing the processor resources among the threads. Such approaches were based on simultaneous multithreading (SMT) [13], often on a single processor core and used the flexibility of instruction scheduling offered by the microarchitecture in superscalar processors. As chip multiprocessors (CMPs) became increasingly ubiquitous, there were studies [8] [9] that demonstrated the viability of executing the redundant threads across separate processor cores.

Studies that seek to mitigate the contention for the core's memory and execution resources amongst the multiple redundant threads advocate the idea of partial redundant multithreading. Such *partial* approaches duplicate only a subset of the dynamic program instruction stream at the cost of lower fault coverage. For example, opportunistic RMT [14] entails execution of the program instructions during phases where the contention for microarchitectural resources are low and turned off when the primary computation needs all the resources. The SliCK approach [15] proposed creation of backward instruction slices, and in concert with prediction

based on value locality, enabled partial RMT only for specific instruction slices.

Software based redundancy approaches typically employ compiler analysis and transformation to duplicate the program code either through operating system visible redundant processes, or through threads within the same process context. This enables more flexibility in terms of selection of instructions to execute redundantly as well as in assigning the redundant execution contexts to hardware resources. The DAFT [16] approach uses a compiler transformation that duplicates the entire program in a redundant thread that trails the main thread and inserts instructions for error checking. The SWIFT [17] and EDDI [18] duplicate instructions and "compare" instructions are inserted to validate the program correctness at appropriate locations in the program code. The ROSE::FTTransform [19] through source-to-source translation duplicates individual source-level statements to detect transient processor faults, but within a single thread context.

VI. Conclusion

As faults become increasingly prevalent in HPC systems, techniques that enable the detection of errors is an important capability for scientific applications. However, when using redundancy, the performance impact of such techniques is a concern. We demonstrated opportunistic application level fault detection where a language-level directive allows application programmers to tailor the extent of fault detection to the features and specific requirements of applications. This framework is supported by a compiler infrastructure and a runtime system. In this paper, we presented an optimization to this approach that offers lazy fault detection. The application's primary computation is prioritized over the duplicate computation by relaxing the requirement that the redundant threads synchronize and compare results following completion of each structured code block. To further support the lazy detection, we also proposed a static thread-to-core assignment strategy that maps all the redundant threads and the detection thread to a dedicated core in a SMP system. While complete redundant execution incurs overheads to application performance to the order of at least 2x, we observe that such a compiler-driven framework supported by runtime adaptation provides more reasonable overheads. The lazy detection approach enables even lower overheads to the HPC application's time to solution. The results promise substantial performance advantages when RMT based fault detection is employed in the context of long-running scientific applications on future extreme scale HPC systems.

Acknowledgment

Partial support for this work was provided by the US Army Research Office (Award W911NF-13-1-0219) and through the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under award number DE-SC0006844.

This work was also supported by the U.S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] J. Dongarra, P. Beckman, T. Moore, and et al., "The International Exascale Software Project Roadmap," *International Journal on High Performance Computing Applications*, pp. 3–60, February 2011.

[2] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward Exascale Resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

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

[4] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithmic Based Fault Tolerance Applied to High Performance Computing," *CoRR*, 2008.

[5] D. Fiala, F. Mueller, C. Engelmann, and et al., "Detection and Correction of Silent Data Corruption for Large-scale High-Performance Computing," in *The International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 78:1–78:12.

[6] S. Hukerikar, P. Diniz, and R. Lucas, "A Case for Adaptive Redundancy for HPC Resilience," in *Euro-Par 2013: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, 2014, pp. 690–697.

[7] S. Hukerikar, K. Teranishi, P. C. Diniz, and R. F. Lucas, "Opportunistic Application-level Fault Detection through Adaptive Redundant Multithreading," in *Proceedings of the International Conference on High Performance Computing & Simulation*, July 2014.

[8] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," *SIGARCH Computer Architecture News*, pp. 99–110, May 2002.

[9] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," *SIGARCH Computer Architecture News*, pp. 25–36, May 2000.

[10] "Rose Compiler," http://www.rosecompiler.org.

[11] P. Sao and R. Vuduc, "Self-stabilizing iterative solvers," in *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2013, pp. 1–8.

[12] T. Slegel, I. Averill, R.M., M. Check, and et. al, "IBM's S/390 G5 Microprocessor Design," *Micro, IEEE*, pp. 12–23, 1999.

[13] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-Fault Recovery using Simultaneous Multithreading," in *29th Annual International Symposium on Computer Architecture, 2002*, 2002, pp. 87–98.

[14] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic transient-fault detection," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, May 2005, pp. 172–183.

[15] A. Parashar, A. Sivasubramaniam, and S. Gurumurthi, "Slick: Slice-based locality exploitation for efficient redundant multithreading," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 95–105.

[16] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, "DAFT: Decoupled Acyclic Fault Tolerance," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10, 2010, pp. 87–98.

[17] G. Reis, J. Chang, N. Vachharajani, and et al., "SWIFT: Software Implemented Fault Tolerance," in *International Symposium on Code Generation and Optimization, 2005*, 2005, pp. 243–254.

[18] N. Oh, P. Shirvani, and E. McCluskey, "Error Detection by Duplicated Instructions in Superscalar processors," *IEEE Transactions on Reliability*, pp. 63–75, March 2002.

[19] J. Lidman, D. Quinlan, C. Liao, and S. McKee, "ROSE::FTTransform - a Source-to-Source Translation Framework for Exascale Fault-tolerance Research," in *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, June 2012, pp. 1–6.