SAND2014-4336C

Opportunistic Application-level Fault Detection through 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*—As the scale and complexity of future High Performance Computing systems continues to grow, the rising frequency of faults and errors and their impact on HPC applications will make it increasingly difficult to accomplish useful computation. Redundancy based approaches usually entail complete replication of the program state or the computation and therefore incur substantial overhead to application performance. Therefore, the wide-scale use of full redundancy in future exascale class computing systems is not a viable solution for error detection and correction.**

In this paper, we present an application level fault detection approach that is based on adaptive redundant multithreading. Through a language level directive, the programmer can define structured code blocks which are executed by multiple threads and the outputs are compared to detect the presence of errors in the computation. The redundant execution is managed by a runtime system which continuously observes and learns about the fault tolerance state of the system resources and reasons whether redundant multithreading should enabled/disabled. The compiler outlines the code blocks and by providing such flexible building blocks for application specific fault detection, our approach makes possible more reasonable performance overheads than full redundancy. Our results show that the overheads to application performance are in the range of 4% to 70% rather than those incurred by complete replication, which are always in the excess of 100%. These are achieved through programmer managed fault coverage together with the runtime system managing redundant execution based on continuous awareness of the rate and source of system faults.

I. Introduction

High Performance Computing (HPC) systems today employ millions of processor and memory chips to drive floating point performance. These recent trends suggest that future exascale HPC systems will be built from hundreds of millions of components organized in complex hierarchies [1] to satiate the demand for faster and more accurate scientific computations. In systems of such scale, reliability becomes a significant concern, since the overall system reliability decreases rapidly with growing number of system components. Long-running scientific applications are therefore increasingly likely to incur errors during their execution on such large systems, limiting application scalability.

The dominant techniques over the past two decades, to deal with failures in HPC systems, have been those based on checkpoint and restart (C/R). This entails an overhead to application performance due to the need to periodically save the complete state of the system to persistent storage. Therefore, such techniques will be increasingly problematic in future massive-scale HPC systems. Some studies even suggest that C/R may no longer be a viable solution if the interval to create and commit coordinated checkpoints or that for recovery exceeds the Mean Time to Failure of the system [2].

Other well-known fault tolerance techniques for error detection and correction include those based on redundancy, either in space or in time. An N-modular redundancy approach entails creating as many replicas of the computation. Errors are detected and in some cases masked by comparing the results and majority voting. HPC applications themselves can participate in the process of fault detection and correction. Much research has been done on Algorithm based Fault Tolerance (ABFT) [3] in the context of various scientific algorithms. These techniques employ encoding on specific data structures and require the application programmer to adapt the algorithms to operate on the encoded data structures. While ABFT techniques offer the application with capabilities to detect and correct errors within its state, there are currently no mechanisms for such application level knowledge, that is unique to specific scientific algorithms, to be exploited by the system software. None of the presently used fault detection and correction techniques recognize that faults in the system are after all anomalous events and therefore do not seek to understand the nature of the fault, its location nor do they assess its impact on the application outcome. Therefore in the case of most current techniques, the trade-off space between the fault coverage offered by a resilience mechanism and the overhead to application performance is not explored.

In this paper, we present an application level solution that adaptively applies redundant multithreading for fault detection. We propose a simple language level directive to allow HPC application programmers to define regions or functions in their code which can be executed by redundant thread contexts for fault detection. Such a language directive then enables our compiler infrastructure to create *flexible* spheres of replication. A sphere of replication is a compiler outlined structured code block that is optionally executed by multiple redundant threads.

Also integral to our approach is a runtime system that monitors events in the system which suggest that faults are imminent. By observing the rate and source of such events, the runtime can make reasoned judgments on when to execute the compiler outlined code blocks using redundant threads and the extent of these spheres of replication. The application level fault detection is therefore enabled opportunistically. Early fault detection at the application level can often enable early and effective recovery or task migration before it leads to catastrophic results for the application's outcome. In this work we focus on observing and learning from warnings for imminent faults and not on the prediction of faults themselves. Our strategy is entirely software based and does not have any specific requirements from the hardware. While the occurrences of faults are inherently unpredictable, there are sufficient indicators in modern hardware systems that can provide early warnings that the system is about to experience more faults/errors. Therefore, the application begins fault detection only when the runtime infers that the system resources are experiencing events that may be construed detrimental to application correctness.

The rest of this paper is organized as follows: Section II describes the concept of redundant multithreading (RMT) while Section III details the need for flexible and adaptive application of RMT for fault detection. Section IV explains the role of the runtime system and the mechanics of the adaptive redundant multithreading for fault detection. Section V describes our experiments while Section VI presents experimental results.

II. Redundant Multithreading

The basis of our approach is Redundant Multithreading (RMT) which has been used as a common form of fault detection [4]. Unlike techniques that use redundant bits on storage and logic components in hardware, RMT duplicates part or complete program execution and compares the respective outputs to detect the presence of errors. One of the key concepts in RMT is the *sphere of replication* [5], which represents a logical boundary that includes an unit of computation that can be replicated (Figure 1). Any fault that occurs within the sphere of replication propagates to its boundary. Faults are detected by comparing specific outputs from the redundant execution of the spheres. Any faults that do not manifest themselves at the boundary of the sphere in the output values get masked and are treated as benign in terms of their impact to the application outcome.

The benefit of software-based RMT approaches is that the redundant execution tends to be loosely coupled. The inputs at the boundary of the sphere of replication and the outputs that are compared tend to be part of the application programmer visible state. Much of the intermediate state does not need to be replicated amongst the redundantly executed contexts. Needless to say there is degradation in application performance due to partial or complete redundant execution of the application's instructions by two independent threads. In the context of scientific applications running on large scale systems, the performance and energy penalty of using program-level RMT at a system-wide level is potentially massive.

While using replication for fault detection and/or correction, managing the size and extent of the sphere of replication is an important consideration that affects the overhead to application performance. The key questions are:

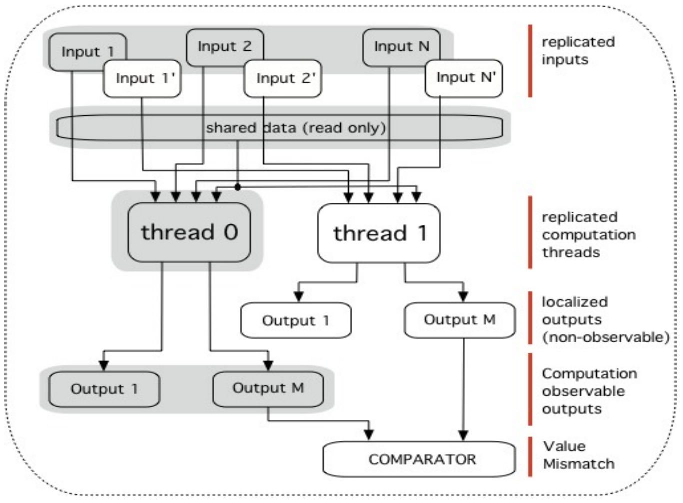

Fig. 1. Concept of Redundant Multithreading (RMT)

- *For which parts of the HPC application will the redundant execution mechanism detect faults?*
  Fault coverage can be extended to specific regions of scientific application code whose execution outcome will indicate the presence of errors in the program state of the application.
- *What are the inputs to the application code block and whether they need to be replicated?*
  Scientific applications often operate on large problem sizes and replicating inputs has important implications for cache bandwidth performance. On the other hand, the failure to replicate inputs may potentially lead the redundant threads on divergent execution paths.
- *What are the outputs values from the application code block that need to be compared in order to detect the presence of faults in the computation?*
  In general, any values that are computed within the sphere of replication must be compared to check for mismatch. The failure to compare meaningful application values potentially compromises fault coverage and the usefulness of redundant execution. However, unnecessary comparisons of output values increases the overhead without improving fault detection coverage.

III. Flexible Spheres of Replication

Various algorithmic fault tolerance techniques are based on application programmers leveraging their scientific domain knowledge to adapt the data structures and algorithms for error detection and correction [3]. Application programmers tend to be well-positioned to understand the fault coverage requirements of their application codes. Often the algorithms of these scientific applications afford partitioning into well-defined regions of computation that map well onto spheres of replication. The programmers can identify such code regions, which may be executed by redundant thread contexts, and specific variables in the application state that can be checked to indicate the presence of errors. However, there are currently no convenient mechanisms that allow programmers to control the fault coverage to match the requirements and algorithmic features of the application. Therefore, to effectively manage the overheads of redundant computation for scientific applications, we propose *flexible* spheres of replication. Based on a preprocessor directive, these offer an application level scoping mechanism that allows the programmer to delineate specific regions in the application code that require fault detection. This constitutes an explicit programming model approach for application level fault detection. It allows the programmer to control the fault coverage in their codes and therefore manage the performance overhead of replicated execution.

A. Robust Directive: Syntax and Semantics

We present a programming directive that is based on a preprocessor pragma directive [6] that enables programmers to define regions of application code. The syntax is as follows:

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

The code that is textually enclosed between the beginning and the end of the code block following the directive is executed by independent threads within the application process. The extent of the directive however does not span multiple functions/routines or code files. The application code contained in the directive scope represents an unit of execution that can be passed to a threading library.

The programming directive allows for the following scoping clauses, similar to those offered by the OpenMP standard:

- `private`: The data variables in this list are treated *private*, that is each redundant thread has its own copy.
- `shared`: The data variables in the *shared* can be accessed by either of the redundant threads and are held in a single memory location.
- `compare`: The data variables in the list are compared at the end of execution of both redundant threads. Any mismatch in the values is communicated to the runtime system.

While the `#pragma robust` directive provides the programmer with control over the scope of specific regions in the application code, it is the compiler which outlines these regions into spheres of replication. Whether these spheres are executed by redundant threads for fault detection is a decision that is deferred to the runtime system. When application execution begins, the code contained in the `#pragma robust` code block programmer is executed serially. When the runtime system signals the application, it enables redundant multithreaded execution. Additionally, whether the inputs should be replicated is also determined by the runtime system. Any mismatch in the values of the variables in the compare clause is communicated to the runtime system. In our implementation of the runtime system, the default behavior is to terminate and restart the application. Such flexibility in the scope of the spheres of replication and dynamic adaptation between serial and redundant multithreaded execution allows opportunistic application level fault detection.

B. Compiler Outlining

We employ source outlining to extract the structured block encompassed by the `#pragma robust` directive and create an outlined function. The outlined function can be executed serially by being called as a regular procedure call. Alternatively, when the runtime signals the application, the function is executed by duplicate threads by passing its pointer to a threading library. Our compiler inserts code such that the outlined function is conditionally executed, either in serial or multithreaded mode, based on a flag that is managed by the runtime. Additionally, the application code is extended, through source level transformations, with instructions that compare the values generated by the redundant threads and informs the runtime system in case of a mismatch in values. The compiler used in this work is based on the ROSE [7] source-to-source compiler infrastructure.

For the data variables that are specified in the private scoping clause, a new object created for each of the redundant threads. The data variables in the shared clause are optionally replicated. To accomplish this, the compiler duplicates the declaration of such variables for each redundant thread.

IV. Runtime Adaptation for Opportunistic Fault Detection

The role of the runtime system in the opportunistic fault detection is to leverage the flexible spheres of replication. This entails enabling/disabling the redundant execution when appropriate. The runtime system also plays a role in determining the scope of the spheres of replication and manages the mapping of the redundant threads to processor cores.

A. System Events and Indicators

The basis of our opportunistic fault detection is the well-reasoned use of redundant execution. This requires that the runtime continuously evaluate the fault tolerance state of the system resources by observing anomalous events in the system. These events usually provide sufficient indication on the future occurrence of errors that are potentially fatal to the applications running on these systems. Broadly, the anomalous events may be classified into three categories: (i) *Uncorrected errors*: these are unrecoverable errors from the hardware point of view. The system software may potentially recover from such errors through software based correction or masking; (ii) *Corrected errors*: these are potentially correctable in hardware, without engaging the system software; (iii) *Fatal errors*: such events are correctable neither in hardware nor by system software. These are usually catastrophic for the system and the applications running on them. Our runtime system monitors and logs events from the first two categories for each processor core and DRAM DIMM module in the system.

The occurrences of such events are communicated up the system stack through an interrupt mechanism. When the interrupt is raised, the operating system logs the error and either kills the application or reboots the system based on the type and location of the error. Modern hardware design specifications include mechanisms to detect and communicate such anomalous event knowledge to the software stack. For example, the Machine Check Exception (MCE) [8] is part of the x86_64 specification. Similarly, the ACPI Platform

Error Interface [9] provides indicators on hardware component status, overheating, bad DIMMs etc. while the Advanced Error Reporting (AER) for the PCI Express gathers and reports the error information. Examples of events are (i) correctable ECC errors on DRAM DIMMs, (ii) PCI bus parity errors, (iii) cache ECC errors, (iv) DRAM scrubbing (which if occurs too frequently indicates a potential DIMM failure) (v) TLB Errors (vi) Memory Controller errors. Based on the type of event and interval between them, the runtime system makes a quantitative assessment of the vulnerability of the system resources.

B. Managing Redundancy and Flexible Spheres of Replication

The critical decision that the runtime system needs to make is when to enable and disable the redundant multithreaded execution of the programmer defined spheres of replication based on the occurrence of the anomalous events. The runtime system defines a metric called the Time Between Events (TBE) (which is equivalent to the Time Between Failure (TBF) metric, except that we are interested in events rather than failures). Similarly the runtime also tracks the Time Since Last Event (TSLE) for each resource it monitors, including the processor cores and DIMM memory modules. Additionally the runtime separately calculates these metrics for cache based events, processor exceptions and for DRAM DIMMs.

When application execution begins, the TBE for all resources is initialized to zero. Any instance of the pragma defined structured code block is executed serially. Upon the occurrence of the first event after execution begins, the runtime enables redundancy through duplicate threading for the execution of all subsequent pragma defined structured code blocks. The output values are compared to detect the presence of errors in the computation.

The TSLE for the system resource that experienced the event is initialized and is updated with the application execution time elapsed since the event. The TBE is also updated as subsequent events occur. When the TSLE exceeds the TBE, the redundant execution is turned off. Further instances of the structured code blocks are executed serially i.e. with fault detection disabled. This enables more effective management of the application performance overhead such that fault detection is enabled over a limited interval following anomalous events, in anticipation of further events. The redundant execution is disabled when no events are experienced over a certain interval and the system resources are deemed *stable*. However, during periods that experience intermittent events or bursts of errors, the fault detection remains enabled for an extended duration.

The other important capability that the runtime system offers is managing the scope of the spheres of replication. While the compiler outlines the pragma defined structured blocks, whether the input variables to the code block are included within the sphere of replication is decided upon at runtime. These decisions are based on the observation of the type of events. Duplicating the inputs places greater bandwidth demand on the memory hierarchy. Therefore, in the case of processor exceptions, only the outlined structured block is executed redundantly without replicating the shared inputs. When the TSLE of cache or DRAM ECC events exceeds the TBE, the runtime also enables the replication of the shared inputs such that the redundant threads have a private copy of the inputs. This enables the runtime system to control the extent of the fault coverage and manage the associated application performance overhead based on the type of events in the system.

C. Resilience Aware Thread Mapping

We can view the sphere of replication as defined by the pragma directive at the application level as a *logical* unit of computation. The code block is executed by multiple redundant threads on processor cores in the *physical* domain. The runtime system extends the fault coverage of the structured code block from the logical to the physical domain by managing the mapping of the redundant threads to processor cores. The runtime system assigns the duplicate threads to specific processor cores in a shared memory multiprocessor (SMP) system by observing the event rates for each processor core.

Our runtime system employs the following two policies:

- *Temporal Redundancy*: based on mapping the redundant thread as a trailing thread behind the main thread on the same processor core. This extends the fault coverage of the computation in time rather than space. It also offers data locality advantages since the shared input data variables can be used by either of the redundant threads. The runtime system employs this policy when the events are transient.
- *Spatial Redundancy*: based on mapping redundant threads to separate processor cores. This provides broader fault coverage since the redundant threads are executed on separate hardware, but needs the input data to be replicated and communicated to the private cache of the core that runs the duplicate redundant thread. The runtime system extends the fault coverage across separate physical cores when the processor executing the main thread has experienced intermittent events or a burst of fault events in a short period of time. The duplicate thread is assigned to a ”healthy” core *i.e.* one that has experienced fewer events in its recent history.

D. Example of Runtime Adaptation for Opportunistic Fault Detection

To illustrate how adaptive redundant multithreading using the above mechanisms enables opportunistic fault detection, we show a simple and illustrative example using the sparse matrix-vector multiplication (SpMV). The SpMV computation consists of iteratively multiplying a constant sparse matrix by an input vector. We assume that the sparse matrix is represented by the Compressed Storage Row (CSR) format which enables better memory access latencies, memory bandwidth and lower cache interference. Although various optimizations are possible based on matrix blocking strategies, for the purpose of this illustration we place the body of the outer for loop i.e. the inner product in the `#pragma robust` structured code block (as shown in the code below). The reduced vector element y[i] for every row is the output value generated at the boundary of the sphere of replication that can be compared when the iteration of the outer loop is executed by duplicate threads.

```
for (i = 0; i < N; i++)
#pragma robust private(j) shared(row_ptr, a, x)
                compare(y[i])
{
   for(j = row_ptr[i]; j < row_ptr[i+1]; j++)
      y[i] += a[j]*x[col_ind[j]];
}
```

The figure 2 illustrates the timeline view of the execution of the SpMV. Each trapezoidal structure represents the execution of a structured code block by duplicate threads and the comparison of the output values to detect errors, before the execution continues. We illustrate two scenarios: Figure 2(a), represents an execution that experiences a single transient event. When the execution commences, the matrix row - vector multiplication is executed serially *i.e.* by a single thread context. When the first event occurs, the interrupt to the OS causes a signal to be passed to the runtime system which in turn causes the runtime to signal the application to enable RMT. The subsequent iterations of the outer loop (i.e. the sparse matrix row-vector multiplication) are executed by the compiler inserted code that forks duplicate threads and compares the reduced y[i] produced by each thread. When the runtime observes no events for an interval longer than TBE (i.e TSLE >TBE), it disables the RMT by passing a signal to the application.

The figure 2(b) illustrates an SpMV execution run during which the runtime observes the occurrence of a single transient event early into the execution and burst of multiple events later on. Since the first fault event occurs very early into the execution, only a limited number of iterations are performed by duplicate redundant threads. However when the second event occurs, it is followed by a burst of events such that the TSLE continuously remains smaller than the TBE. The RMT execution is not disabled until the application converges. If there is mismatch in the output value y[i] during any iteration, the runtime is notified. The runtime can initiate recovery, task migration or terminate and restart the application. But since the emphasis of this work is on fault detection, the current runtime system design always chooses to terminate the application.

V. Experimental Evaluation

A. Fault Model

The adaptive RMT is based on the observation of anomalous events in the system. Our fault injection framework simulates such fault events and the resulting interrupt by signaling the runtime system. This is accomplished by sending the USR1 signal to the runtime system which in turn signals the application process to enable or disable the redundant execution of the compiler outlined spheres of replication. The framework simulates events such as corrected DRAM ECC error notifications, PCI bus parity errors, corrected cache ECC errors, DRAM scrubbing notifications and recoverable processor exceptions. The nature of the fault events is such that they do not perturb any aspect of the application state or the computation. The fault events are randomly generated during the application execution and are logged by the runtime.

B. Applications

We evaluate four computational kernels/applications that demonstrate the application level fault detection through the creation of flexible spheres of replication and adaptive RMT.

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 y[i] 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 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 [10] 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 stability condition at the end of each iteration.

C. Methodology

Through programmer specified fault coverage which is adaptively managed by a runtime system, we seek to accomplish fault detection at more reasonable overheads to application performance than with complete redundancy approaches. Therefore to understand the impact of adaptive redundant multithreading on the application performance, we first study the overheads in the case of single threaded versions of each of the application codes described above. 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. The runtime determines whether and when to enable or disable the redundant execution based on the TSLE and TBE metrics.

We also seek to understand the performance impact of adaptive RMT based fault detection in the context of the multithreaded implementations of the above application codes. With recent the trend in HPC system architectures of building

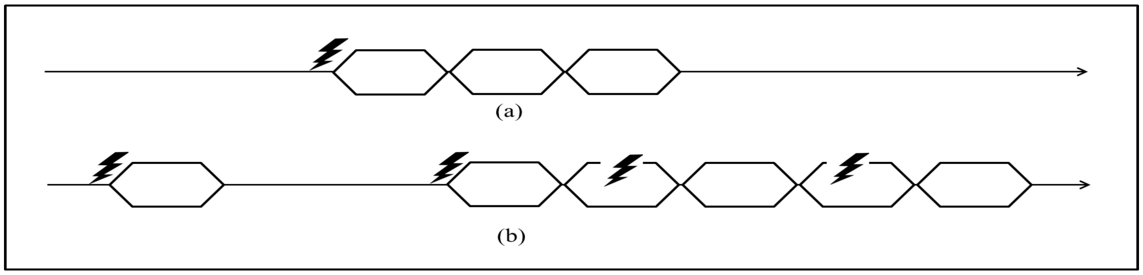

Fig. 2. Example of Runtime Adaptation for Opportunistic Fault Detection: Timeline View of SpMV Execution

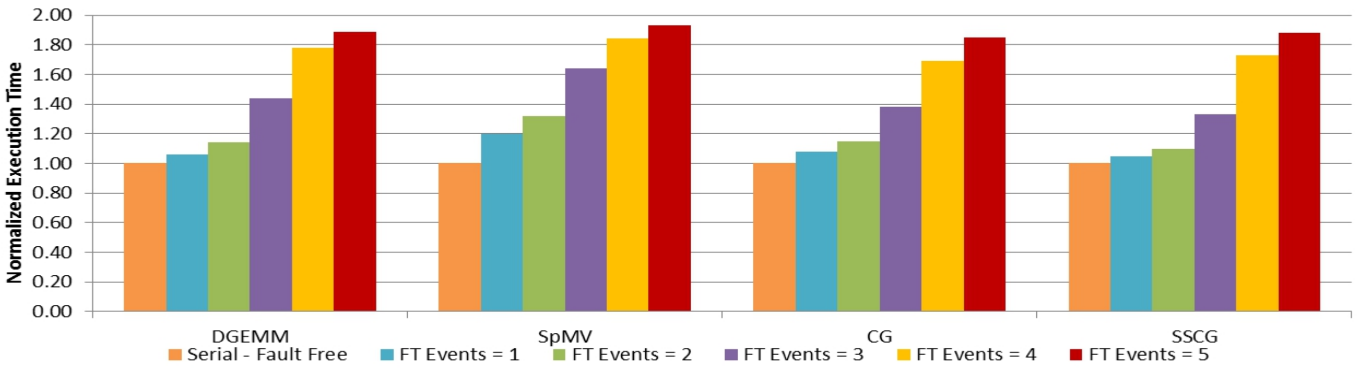

Fig. 3. Results: Performance Evaluation of Fault Detection with Adaptive Redundant Multithreading

systems with multiple cores per chip and multiple sockets per compute node board, applications use explicit shared memory programming paradigms such as OpenMP, pThreads to distribute the computation across all the cores in the system. The adaptive RMT accounts for the source of the events and redundant execution is enabled only on processor cores which have experienced events recently. The healthy cores *i.e.* those which have not seen any event in the recent past continue executing their programmer defined blocks in sequential mode with no fault detection. For these experiments, we also perform random fault event generation and evaluate the overhead to the application's time to solution, when the RMT is enabled only for specific processor cores.

We evaluate the thread assignment policies where the runtime system manages the mapping of redundant threads to separate processor cores (spatial redundancy policy) or as trailing thread to the same processor core (temporal redundancy policy), based on the frequency of fault events. For each fault event rate, we perform 10000 application runs with randomized event generation to measure the normalized average execution time with reference to a fault free execution time. The evaluation platform is an Intel ^{TM}Xeon 8-core 2.4 GHz compute node running the Linux operating system in the USC HPCC cluster.

VI. Results

The Figure 3 shows average normalized execution times for the serial implementations of each of the application codes. The baseline is a fault free execution run (represented by the 1.0x data points in Figures 3 and 4). The remaining data points show the average normalized application execution times (for the 10000 experiment runs) when a count of 1, 2, 3, 4 and 5 random fault events are generated during each application run. In the case of DGEMM, the occurrence of a single event adds only 5% overhead to the application performance since a limited number of iterations of the dot product computation are executed by a redundant thread with checking for errors. The overhead for SpMV is higher at 20% while the CG and Self Stabilizing CG is at 8.5% and 7% respectively. As the number of events per run increases, the amount of computation that is performed redundantly increases quickly because the runtime tends to behave conservatively by leaving the redundant execution and output checking enabled when there is a burst of fault events or if they occur intermittently. However, even in the case of five randomized fault events per application run, which leads to a large chunk of the iterations of DGEMM, SpMV, CG, SSCG to be executed redundantly, the average execution times are 1.78x, 1.81x, 1.7x and 1.74x respectively (in comparison to the serial fault-free case).

The results summarized in Figure 4(a) show the average normalized execution times when the applications use an explicit shared memory programming model such as OpenMP and all the cores on the test platform are put to work. The runtime also logs the location of the events. When fault events are localized to a single processor core, fault detection is enabled for only that specific core. Therefore, the spheres of replication assigned to the core experiencing fault events are executed by a redundant thread and its outputs checked for errors. The remaining computation assigned to the *healthy* cores continues in serial mode. The benefit of such a reasoned utilization of redundancy for fault detection that is cognizant of fault sources is evident in Figure 4(a), which in comparison to Figure 3 shows lower overhead to application execution time for each of the fault rates.

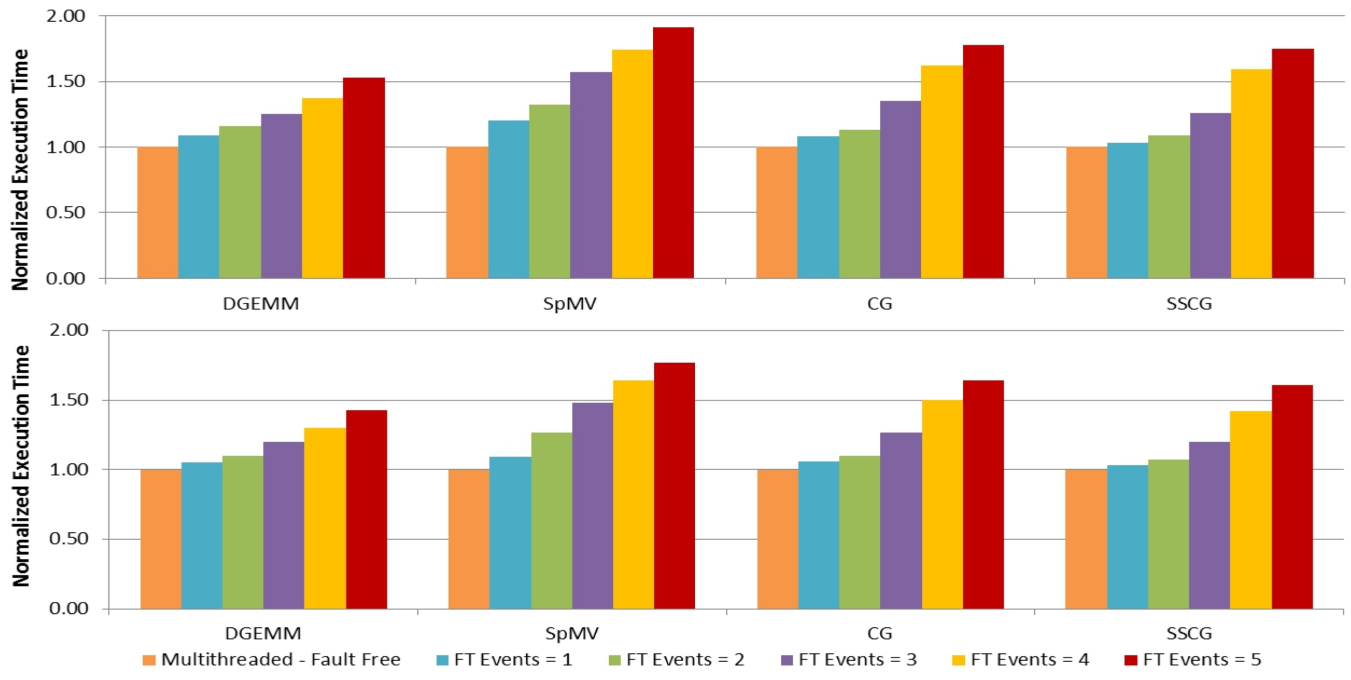

Fig. 4. Results: (a) Multithreaded Computation (b) Core Assignment for Redundant threads to Separate Processor Cores

Figure 4(b) highlights the comparison between the two core assignment strategies described in Section IV-C. While figure 4(a) uses trailing redundant threads, those in figure 4(b) map the redundant threads to separate cores in the context of a SMP system. The former has cache locality advantages since the input data in the private caches is reused, whereas the latter provides more complete fault coverage since the redundant threads are executed on separate hardware. Notwithstanding the need for data to be communicated to the private cache of the core that runs the duplicate redundant thread from the core that runs the original thread, we note that the separate core-mapped redundant thread has marginally lower overhead in comparison to the trailing redundant thread for all the test applications.

VII. Related Work

In HPC systems, studies have argued for multi-modular redundancy in compute nodes and have shown to accommodate a reduction in individual component reliability by a factor of 100-100,000 to justify the 2x or 3x increase in cost and energy [11]. Ferreira *et. al* [12] evaluate the costs and benefits of using MPI process replication as an alternative to the widely used checkpoint restart protocols, while Stearley *et. al* [13] observe that partial process replication helps increase the Job Mean Time to Interrupt (JMTTI) of tasks, but that there is no alternative to full process replication for highly resilient operation. However, given the scale of future exascale systems in terms of number components and the complexity of applications, complete node-level or process-level multi-modular redundancy would incur exorbitant overhead to costs, performance and energy.

Hardware solutions that employ redundancy are transparent to the supervisor software and application programmer, but require specialized hardware. In the domain of commercial transaction processing, fault tolerant servers such as the Tandem Non-Stop [14] and later the HP NonStop [15] used two redundant processors running in locked step. The overheads of energy and cost to widely employ such hardware techniques for exascale HPC systems however, would be extremely high. Approaches that leverage multiple contexts in Simultaneous Multithreaded (SMT) processors have also been studied. Such RMT approaches show slightly lower power and performance overheads in comparison to redundant locked-step processor based systems [5] [16].

Software-based redundant multithreading approaches tend to offer more flexibility and are less expensive in terms of silicon area as well as chip development and verification costs. SWIFT [17] is a compiler-based transformation which duplicates all program instructions and inserts comparison instructions during code generation and the duplicated instructions fill the scheduling slack. The DAFT [18] 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.

VIII. Conclusion

As faults become increasingly prevalent in HPC systems, techniques which can tailor the extent of protection to the requirements of the application and to the state of the system will be needed. In this paper, we presented an approach that enables opportunistic fault detection within the application program state based on a language level directive. It enables the programmer to define structured blocks within their application codes. The error detection via redundant multithreaded execution of such code blocks is opportunistically enabled by the runtime system based on the observation and assessment of the rate and source of fault events in the system. While a complete redundant execution incurs overheads to application performance of the order of at least 2x, such a flexible approach enables more reasonable overheads, in the range of 1.04x to 1.7x. By enabling programmer directed, application specific fault coverage, this approach provides substantial savings to application performance overheads in the context of long-running scientific applications when compared to fault detection based on complete redundant execution.

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] P. Kogge, K. Bergman, S. Borkar, and et al., "Exascale Computing Study: Technology Challenges in Achieving Exascale systems," DARPA, Tech. Rep., Sept 2008.

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

[4] 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.

[5] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *SIGARCH Computer Architecture News*, pp. 25–36, May 2000.

[6] S. Hukerikar, P. Diniz, and R. Lucas, "A Case for Adaptive Redundancy for HPC Resilience," pp. 690–697, 2013.

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

[8] Intel, "Intel 64 and IA-32 Architectures Software Developers Manual," Tech. Rep., 2011.

[9] P. Sakthikumar and V. J. Zimmer, "A Tour beyond BIOS Implementing the ACPI Platform Error Interface with the Unified Extensible Firmware Interface," Intel Corporation, Tech. Rep., January 2013.

[10] 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.

[11] C. Engelmann, H. H. Ong, and S. L. Scott, "The Case for Modular Redundancy in Large-scale High Performance Computing Systems," in *International Conference on Parallel and Distributed Computing and Networks*, February 2009, pp. 189–194.

[12] K. Ferreira, J. Stearley, J. H. Laros, III, and et al., "Evaluating the Viability of Process Replication Reliability for Exascale Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[13] J. Stearley, K. Ferreira, D. Robinson, and et al., "Does Partial Replication Pay-off?" in *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2012.

[14] D. McEvoy, "The Architecture of Tandem's NonStop System," in *Proceedings of the ACM '81 conference*. New York, NY, USA: ACM, 1981.

[15] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop Advanced Architecture," in *International Conference on Dependable Systems and Networks*, 2005, pp. 12–21.

[16] 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.

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

[18] 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.