A Case for Adaptive Redundancy for HPC Resilience*

Saurabh Hukerikar, Pedro C. Diniz, and Robert F. Lucas

Information Sciences Institute,
University of Southern California,
Marina del Rey, CA 90292, USA
{saurabh,pedro,rflucas}@isi.edu

Abstract. Redundancy both in space and time has been widely used to detect and in some cases correct errors in High Performance Computing (HPC) systems. With the HPC community seeking exascale class supercomputers by the end of the decade, unrealistic expectations for correct system behavior will result in exorbitant costs in terms of performance lost and energy expended. Resilience strategies will need to find balance between fault coverage and the overheads incurred. In this work, we propose an adaptive approach that factors in application level knowledge together with runtime inference about the fault tolerance state of the system to dynamically enable redundant multithreading (RMT). Our approach is based on simple programming language extensions, tightly integrated with a compiler infrastructure and a runtime framework that enables managing the performance overheads of redundant computation.

1 Introduction

In the era of exascale supercomputers, systems will frequently encounter faults that lead to system errors and failures. With smaller transistor feature sizes and lower supply voltages in each new process generation, chips are projected to become increasingly vulnerable to errors [1]. Single Event Upsets (SEUs) that cause soft errors are an important class of errors and with shrinking transistor geometries, the likelihood that these result in multi-cell upsets (MCU) will also increase. Error correcting codes (ECC) in memory and parity in latches and registers have been used for detection and in some cases correction so that the errors remain transparent to the application and system software. Exascale High Performance Computing (HPC) systems are projected to deploy millions of processor cores and memory chips organized in complex hierarchies [2]. Even if individual component reliabilities were to remain the same as they are today,

* This research has been supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (Award Number DE-SC0006844). Partial support for this work was also provided through a contract from Sandia National Laboratories (Award Number 1315083).

the sheer scale of these systems would make the Mean Time to Interrupt (MTTI) so small that a significant fraction of the applications' execution times would be spent on failure recovery actions.

Redundancy, either in space or in time, has been widely used for error detection and recovery. An N-modular redundancy approach entails creating as many replicas of the computation whose results are compared to verify correctness and errors are masked through majority voting. In HPC systems, software managed system level approaches that use complete replication at the process level have been shown to reduce the number of interrupts that an application experiences [3]. Redundant multithreading (RMT) involves running identical copies of the same program as independent threads. When there is an output mismatch, the checker flags an error and initiates a recovery sequence. Redundant computation constitutes a preemptive approach to fault tolerance and there is a significant performance degradation associated with using it for entire programs. In future exascale class supercomputing systems, there will be billions of processes and threads, and applying redundancy systemwide for error detection and masking will result in massive penalties to application performance and energy.

We believe that redundancy should be judiciously applied, factoring in the requirements of the application as well as the fault tolerance state of the system. In this paper, we propose an approach that allows the selective use of multithreaded redundancy based on the programmer's insight. The approach is based on modest programming model extensions that allows programmers to specify what regions of the application code require robust computation. Additionally, whether to enable the redundant multithreading is based on monitoring and inference of fault events by a runtime system. This selective and adaptive use of redundancy seeks to minimize the large performance overheads associated with complete replication. We present an initial framework and preliminary results, and discuss future research directions for future exascale systems.

The rest of the paper is organized as follows: Section 2 looks at various redundancy based fault tolerance approaches in hardware and software and makes the case for adaptive redundancy. Section 3 explains the programming construct, its syntax and semantics. Section 4 describes the role of the runtime, and Section 5 presents the experimental evaluation and some initial results while Section 6 describes the ongoing and future work.

2 Background

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 [4]. Ferreira *et. al* [5] evaluate the costs and benefits of using MPI process replication as an alternative to the widely used checkpoint restart protocols, while Stearley *et. al* [6] 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 [7] and later the HP NonStop [8] used two redundant processors running in locked step. Later the IBM G5 [9] employed two fully duplicated lock-step pipelines to enable low-latency detection and rapid recovery. 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 [10] [11].

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 [12] 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 [13] 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 primary limitation of complete replication approaches is that they indulge in preemptive robustness and therefore they account for neither the type of faults, their location and frequency, nor the requirements of the application code and any algorithm specific fault tolerance features of the application.

3 Programming Language Extensions

Many classes of algorithms used in scientific applications selectively require robust computation. However, there are no convenient mechanisms for the programmer to partition the application code into such regions and for the system to utilize such knowledge. For example the Fault Tolerant Generalized Minimal Residual (FTGMRES) algorithm [14] can be partitioned into reliable and unreliable phases. Robust computation is required of only the reliable phases for eventual convergence to the right solution. Several iterative linear solvers [15] and multigrid solvers [16] contain phases that tend to show different fault resilience features. Rather than complete program level redundancy, only the code contained within these programmer defined regions can be executed by multiple redundant threads.

We present a programming model extension that is based on a preprocessor pragma directive that enables delineating code into regions that require robust computation. The compiler inserts the code that enables such regions to be run by multiple independent threads and to compare their respective outputs. The syntax is depicted below:

```
#pragma robust shared<variable list ...>
{
  /*
   code
   ...  */
}
```

The language extension provides OpenMP style data scoping clauses: *shared* to specify the global variables that the code section may consume as input or output; and a *private* clause (not shown above) to list variables whose scope is limited to the section. The compiler used in this work is based on the ROSE [17] source-to-source compiler infrastructure. We use source outlining to extract the code section encompassed by the `#pragma robust` directive to create an outlined function whose pointer can be passed to a threading library. We also extend the application code through source level transformations that insert instructions that compare the values of the sentinel value generated by the redundant threads and informs the runtime system in case of a mismatch in values. In this initial implementation, we also create a duplicate copy of the global data that the section modifies to avoid data races and synchronization issues between the redundant threads.

4 Runtime System

4.1 Runtime Adaptation

While the programmer driven partitioning described in Section 3 seeks to manage the application performance overhead to some extent, most redundancy approaches also do not take into account the fault tolerance state of the system which may alter due to external environmental factors as well as operational features of the system. Soft errors are random external events attributed to alpha particles that originate from within the chip packages and high-energy neutrons from cosmic radiation. Power management techniques that dynamically scale the voltage and frequency also impact the reliability of circuits [18]. Faults attributed to silicon aging that emerge due to temperature induced stresses and other ambient conditions tend to manifest themselves gradually. The occurrence of such faults may be distributed in bursts or as random individual events and therefore the robustness offered by redundant execution would be best served when the system state is vulnerable.

Our runtime system *adapts* to the changing fault tolerance state of the underlying system by enabling RMT execution when the fault tolerance state degrades, for example, if the system experiences too many ECC errors in the caches, or if an application level algorithm based fault tolerance (ABFT) method detects errors once too often. The runtime is implemented as an independent light-weight process that monitors the state of the application and interfaces with the operating system through an interrupt handler mechanism. The runtime maintains a log of fault events in the system. It also maintains a Dynamic Resilience Map

(DRM) [19] which contains the mapping of program-level data structures, functions and code segments, their program-level address offsets, and the corresponding mapping in physical memory, so that it can reason about the significance of the location of the error in the context of the application's logical address space.

For this initial implementation of the runtime system, the notion of a fault event is generalized and may represent a corrected ECC error in the memory, a processor exception or even an error detected by an application level algorithmic fault tolerance method. Also, the current implementation uses a threshold value of event count when deciding whether to initiate the redundant multithreading. The development of better heuristics that base the decision of whether to enable and subsequently disable redundant multithreading on the type and location of the fault, whether the event occurred as an isolated event, or shows a pattern over a period of time, is the subject of ongoing work.

4.2 Core Mapping Policies

The runtime system also makes possible spatial redundancy such that it can leverage multiple processor cores in the context of shared memory multiprocessor (SMP) by assigning the redundant threads to separate processor cores. We explore two policies:

– Adaptive RMT - Trailing Thread: This policy executes the duplicate thread on the same processor core so that the input data variables can be shared by both threads and therefore has some data locality advantages.
– Adaptive RMT - Core Mapping: This policy maps the RMT threads on a separate core in a SMP. This provides more complete 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 can therefore make a reasoned trade-off between the fault coverage requirement and performance overhead. For this implementation the policy for core affinity of the redundant threads is configured at initialization and we are currently exploring heuristics that can dynamically select between the two policies based on type and frequency of fault events.

5 Experimental Evaluation

5.1 Methodology

We analyze the impact of adaptive redundant multithreading approach with DGEMM, a double-precision matrix-matrix multiplication kernel that is a basic building block for dense matrix linear algebra. We utilize the partitioning based on the `#pragma robust` directive described in Section 3 on the inner product of DGEMM. We evaluate application performance to understand the benefits achieved from the programmer guided selective redundancy and the runtime based adaptive enabling of RMT for the following cases:

– Baseline: Process level redundancy where the program is executed by independent operating system created processes.
– Complete RMT - Trailing Thread : The runtime is configured so that redundant multithreading is enabled throughout the application run but only the `#pragma robust` annotated sections are executed by a redundant thread that trails the main thread on the same processor core.
– Complete RMT - Core Mapped Thread : A redundant thread is also created for all annotated sections, but the runtime maps the redundant threads to different cores.
– Adaptive RMT - Trailing Thread: The redundant multithreading is enabled dynamically by the runtime upon occurrence of fault events and all subsequent `#pragma robust` annotated sections are executed by a trailing thread on the same processor core.
– Adaptive RMT - Core Mapping: RMT is also enabled based on runtime inference but the redundant threads are mapped to separate processor cores.

The objective of the complete RMT experiments is to evaluate the performance benefit of selective application of redundancy to only the inner product of the DGEMM rather than the complete program. The adaptive RMT configurations evaluate performance benefit of turning on robustness only when the system experiences fault events. Our fault injection framework allows injection of faults at any point in the application execution and these are logged by the runtime. We generate a single fault event per application run and the instant of event generation is randomized across experiment runs and we perform 10000 runs per matrix problem size. For these experiments, the runtime system is configured such that the threshold for switching the application into redundant multithreading mode is a single error event. The evaluation platform is an Intel $^{\text{TM}}$Xeon 8-core 2.4 GHz compute node running the Linux operating system.

5.2 Results

The figure 1 shows the average execution time for each configuration. The baseline is the performance of process level redundancy where the entire program is executed twice and the final results compared. When redundancy is applied to only programmer annotated regions, but *all* sections in the DGEMM are redundantly executed, the average execution time (of all problem sizes) is 85% of the baseline time for the trailing thread configuration. Further benefit is realized when the redundant thread is assigned to a separate processor core and the average execution time is 25% lower than the baseline and shows much as 42% lower execution time for the largest matrix problem size. For the runtime adaptation, the performance results reflect the average execution time across the 10000 experimental runs. For these runs redundant multithreading is enabled only upon detecting a fault event by the runtime system and the average execution time for all problem sizes is 65% of the baseline execution time with trailing redundant threads. When the dynamically enabled redundant threads are mapped to separate cores, the average execution time is 41% faster than baseline and as much as 52% faster than baseline for the largest problem size.

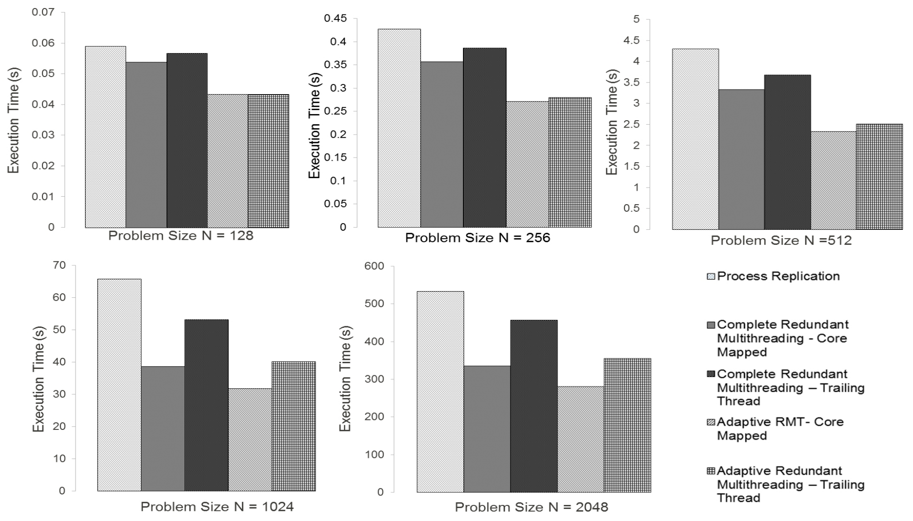

Fig. 1. Results: Performance Comparison of Adaptive Redundant Multithreading

6 Conclusion and Future Work

Redundancy is an effective strategy in the detection and in some cases correction of errors but incurs extremely high overheads. In this paper we presented a software based adaptive application of redundant multithreading that allows for a balanced and reasoned trade-off between application resiliency and performance overheads incurred. Redundancy is enabled based on runtime inference of the system's fault tolerance state and through a simple language extension the programmer's requirements for robustness are incorporated. The initial results suggest that the approach holds promise in managing the trade-off between performance and fault coverage, but much work remains to be done. We are currently broadening the definition of events to include more fault models and fault distribution patterns that are relevant to future exascale systems. This will allow design of effective runtime heuristics that will intelligently enable and eventually disable RMT as well as guide core assignment of redundant threads.

References

1. Shivakumar, P., Kistler, M., Keckler, S., Burger, D., Alvisi, L.: Modeling the effect of technology trends on the soft error rate of combinational logic. In: International Conference on Dependable Systems and Networks, pp. 389–398 (2002)
2. Kogge, P., Bergman, K., Borkar, S., et al.: Exascale Computing Study: Technology Challenges in Achieving Exascale systems. Technical report, DARPA (September 2008)

3. Riesen, R., Ferreira, K., Stearley, J., et al.: Redundant computing for exascale systems. Technical report, Sandia National Laboratories (December 2010)
4. Engelmann, C., Ong, H.H., Scott, S.L.: The Case for Modular Redundancy in Large-scale High Performance Computing Systems. In: International Conference on Parallel and Distributed Computing and Networks, pp. 189–194 (February 2009)
5. Ferreira, K., Stearley, J., Laros III, J.H.: 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, pp. 1–12 (2011)
6. Stearley, J., Ferreira, K., Robinson, D., et al.: Does Partial Replication Pay off? In: IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops, DSN-W (2012)
7. McEvoy, D.: The architecture of tandem's nonstop system. In: Proceedings of the ACM 1981 Conference. ACM, New York (1981)
8. Bernick, D., Bruckert, B., Vigna, P., Garcia, D., Jardine, R., Klecka, J., Smullen, J.: NonStop Advanced Architecture. In: International Conference on Dependable Systems and Networks, pp. 12–21 (2005)
9. Slegel, T., Averill III, R.M., Check, M., et al.: IBM's S/390 G5 Microprocessor Design. Micro, pp. 12–23. IEEE (1999)
10. Reinhardt, S.K., Mukherjee, S.S.: Transient fault detection via simultaneous multithreading. SIGARCH Computer Architecture News, 25–36 (May 2000)
11. Vijaykumar, T., Pomeranz, I., Cheng, K.: Transient-Fault Recovery using Simultaneous Multithreading. In: 29th Annual International Symposium on Computer Architecture, pp. 87–98 (2002)
12. Reis, G., Chang, J., Vachharajani, N., Rangan, R., August, D.: SWIFT: Software Implemented Fault Tolerance. In: International Symposium on Code Generation and Optimization, pp. 243–254 (2005)
13. Zhang, Y., Lee, J.W., Johnson, N.P., August, D.I.: DAFT: Decoupled Acyclic Fault Tolerance. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, pp. 87–98 (2010)
14. Bridges, P.G., Ferreira, K.B., Heroux, M.A., Hoemmen, M.: Fault-tolerant linear solvers via selective reliability. CoRR (2012)
15. Bronevetsky, G., de Supinski, B.: Soft Error Vulnerability of Iterative Linear Algebra Methods. In: Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, New York, NY, USA, pp. 155–164 (2008)
16. Casas, M., de Supinski, B.R., Bronevetsky, G., Schulz, M.: Fault resilience of the algebraic multi-grid solver. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, New York, NY, USA, pp. 91–100 (2012)
17. Rose Compiler, `http://www.rosecompiler.org`
18. Melhem, R., Mosse, D., Elnozahy, E.: The interplay of power management and fault recovery in real-time systems. IEEE Transactions on Computers 217–231
19. Hukerikar, S., Diniz, P.C., Lucas, R.F.: A Programming Model for Resilience in Extreme Scale Computing. In: IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops, DSN-W (2012)