Using Randomization to Cope with Circuit Uncertainty

Hamid Safizadeh*, Mohammad Tahghighi†, Ehsan K.Ardestani‡, Gholamhossein Tavasoli† and Kia Bazargan†§
*School of Computer Science, Institute for Research in Fundamental Sciences, Tehran, Iran, Email: hamid@ipm.ir
†ECE Dept, Isfahan University of Technology, Isfahan, Iran, Email: {m.tahghighi, gh.tavasoli}@ec.iut.ac.ir
‡CE Dept, University of California Santa Cruz, Santa Cruz, CA 95064, USA, Email: eka@soe.ucsc.edu
§ECE Dept, University of Minnesota, Minneapolis, MN 55455, USA, Email: kia@umn.edu

***Abstract*—Future computing systems will feature many cores that run fast, but might show more faults compared to existing CMOS technologies. New software methodologies must be adopted to utilize communication bandwidth and the computational power of few slow, reliable cores that could be employed in such systems to verify the results of the fast, faulty cores. Employing the traditional Triple Module Redundancy (TMR) at core instruction level would not be as effective due to its blind replication of computations. We propose two software development methods that utilize what we call Smart TMR (STMR) and fingerprinting to statistically monitor the results of computations and selectively replicate computations that exhibit faults. Experimental results show significant speedup and reliability improvement over traditional TMR approaches.**

I. Introduction

The end of Moore's Law for silicon-based systems is in sight [1]. We will soon witness the integration of nano-devices such as carbon nano-tube based transistors and wires, quantum dots, spintronics and molecular devices into silicon systems, and even possibly total replacement of silicon circuits by these new technologies. Self-assembly techniques will probably replace or augment lithographic fabrication processes to integrate orders of magnitude larger systems [2].

In recent years, we have been forced to model variations in the behavior of devices to avoid large failure rates and high manufacturing costs [3]–[9]. Device variations and failure rates are expected to increase in future technologies to the point that unreliability of computing systems will create major challenges in further scaling of such systems. No matter what technology (e.g., carbon nano-tubes or molecular devices) emerges as the dominant technology to carry Moore's law forward, future computing systems will have many computational nodes and communication links with high error rates.

We will soon be at a point that such error rates have to be treated as first-class citizens in designing electronic systems, much like the way we model power, area and performance in today's systems [10]–[12]. Using a large number of computing elements and replicating computations both in space (parallelism) and time (multiple runs of the algorithm on the same subset of computing elements) will help us reduce the errors in the final solutions of the problems that need to be solved on these inherently unreliable machines. Existing fault tolerant methodologies do not consider the whole range of programming abstraction from algorithms to devices. Techniques such as Triple Module Redundancy (TMR) would not scale well because as the system becomes larger and the fault rates increase, more defects will appear on the sub-systems to the point that the replication of resources would show no advantage over the original system. The authors in [13] showed that replicating resources on an FPGA using future technology models will be useful only to a certain point, after which the added area will incur additional faults and render redundancy useless. Smarter, more selective methodologies have to be adopted that can continue the Moore's law in the near and long-term future.

In this paper, we propose a new programming model that uses concepts from the field of randomized algorithms and requires programmers to address parallelism and reliability explicitly. The underlying architecture consists of a robust processor (e.g., implemented in CMOS) connected to a host of faulty processors (future CMOS or more fancy technologies). For any given algorithm, the programmers are asked to specify how the algorithm is partitioned across the faulty processors, how the results are to be verified and combined by the robust processor and how erroneous operations should be dispatched to faulty processors again for re-computation. The key point is that the verification algorithms must be much faster than the actual computations because presumably the robust processor is much slower than the faulty ones, and that is where the randomization would be useful.

The rest of the paper is organized as follows. Section II describes the underlying assumptions. Section III covers background material for our verification mechanisms. In Section IV, we list different applications that we have implemented to test the effectiveness of our proposed methods. Experimental results are presented in Section V and the paper is concluded in Section VI.

II. Architecture

In this section, we will outline our assumptions about the architecture and its fault models. We will also describe our proposed parallelism and fault tolerance software methods that we will employ in our experiments. Fig. 1 shows

978-3-9810801-5-5/DATE09 © 2009 EDAA

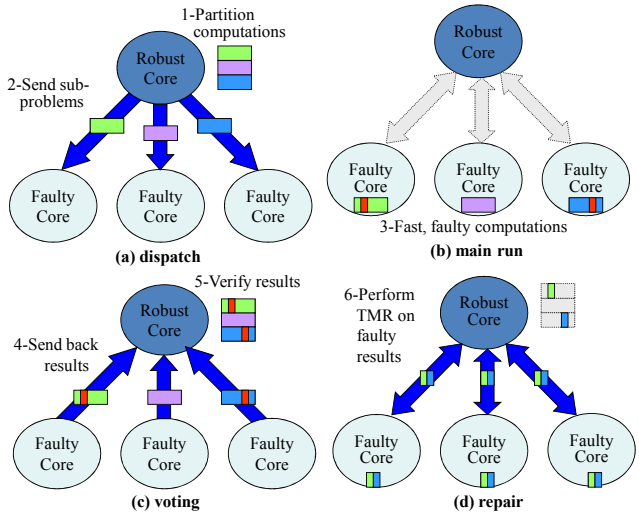


Fig. 1. Example of how our fingerprinting methodology works. (STMR is similar but NOT the same). (a) the robust processor divides computations among fast, faulty processors. (b) computations are performed in parallel. (c) results are sent back and verified by the slow, robust processor. (d) computations that were faulty are sent to be "repaired" using re-computation and/or TMR.

the architecture in which a robust processor is connected to three faulty processors. We use the terms core and processor interchangeably. In our experiments in Section V, we will use up to nineteen faulty processors that are connected to the robust processor.

We assume that the robust processor is fabricated using a technology similar to today's technology in which fault rates can be brought down very close to zero using circuit techniques. On the other hand, faulty cores are built using future technologies. These processors are much faster than the robust processor but might exhibit permanent or transient faults. In this study, we aggregate computation, memory and communication faults into a single probability that an atomic operation (e.g., a scalar multiplication) generates a faulty output. A more detailed model in which memory, communication and computation faults are modeled differently could easily be derived from our existing models.

III. Programming Model

Our proposed methodology requires programmers to follow a template, similar to what object-oriented programs require. In OOP, programmers are required to specify public and private data members and methods, object construction and destructions, exceptions, etc. At a higher level of abstraction, we require that the programmers specify:

1) Dispatch: how the computations are to be divided and dispatched to the processors. Inner and outer loops of algorithms, in addition to when data gather and synchronization should be performed are specified here. Fig. 1(a) shows this step.

2) Main run: how individual faulty processors should perform computations (Fig. 1(b)). This is where the bulk of computations take place. Depending on the verification method employed, programmers might use the faulty processors to generate some form of signature (similar to CRC error-checking codes) to be later used by the robust processor for a light-weight voting. We will address this issue in more detailed in the next subsections (STMR and fingerprinting).

3) Voting: how the robust processor should receive signatures and run a light-weight verification algorithm to determine if the results are correct (Fig. 1(c)).

4) Repair: finally, how unresolved faulty operations should be handled, e.g., be sent for re-computation (Fig. 1(d)). In this paper, we assume that TMR is also used for this step on parts of the data that exhibited faults.

For the verification process, we propose two methods and compare their effectiveness to the traditional TMR: one is STMR, and the other is fingerprinting. The next two subsections describe these methods in more detail.

A. STMR

Similarly to the traditional TMR method that is applicable to every application, this approach first broadcasts all the data of size R to all N processors ($N=3$ in Fig. 1. Note that in the figure, data is divided between the processors but in STMR the whole data is broadcast to all). But unlike TMR, it requires the N processors to locally compute a signature and send that for voting / verification. A signature is a vector of size R/K, where each of its components is generated by compressing (XORing) K consecutive elements of the output (called a "chunk" of data) into one number. We also call K the "Partition Size". Essentially generating a signature is similar to down-sampling the output.

This method is faster than TMR for small fault rates because only $N*R/K$ elements need to be sent back in the *voting* step to the robust processor for voting. If a majority of signatures for a chunk of data agree, then the processor needs to read the chunk from only one of the agreeing faulty processors (as opposed to N in TMR). If the signature of a chunk does not pass successful voting, that chunk of data (i.e. K elements) will be dispatched for traditional TMR in the *repair* step.

For small values of the atomic fault rate ρ, the larger the K, the faster is the voting time (both communication and voting), but the slower is the repair step because the chunk of data that is sent for re-computation is larger. On the other hand, if ρ increases, many of the chunk signatures will fail voting and the whole process of generating signatures becomes redundant and causes a runtime increase compared to TMR. To strike a balance between these factors, K should be proportional to $1/\rho$. See the appendix section for more details.

B. Fingerprinting

The term "Randomized Algorithms" refers to the class of algorithms that trade-off runtime (resources) with the accuracy of results. Among the techniques used in randomized algorithms is *fingerprinting*.

Fingerprinting refers to the process of selecting subsets from two sets P and Q to check whether $P=Q$ or not [14]. The subsets are called "fingerprints" and are much easier to check for equality than the original problems. There may be many ways that a fingerprint can be generated for a given problem. Choosing many random fingerprints will increase the probability of correctly identifying whether $P=Q$ or not. An example of fingerprinting is the Freivalds algorithm [15], which checks the correctness of large matrix multiplications. Assuming that an unreliable machine has computed $C=A\times B$ (A, B and C are large $n\times n$ matrices), the Freivalds method randomly chooses a vector $\alpha\in\{0,1\}^n$ and computes $\beta=A.(B.\alpha)$ and $\gamma=C.\alpha$. If $\beta\neq\gamma$, output "$C\neq A\times B$" (we have implemented a modified version of the Freivalds algorithm. See Section IV). If f fingerprints are generated, the probability of not catching the error is at most $1/2^f$. Note that checking one fingerprint would take $O(n^2)$, as opposed to $O(n^3)$ of a direct method that re-computes $A\times B$ and compares the result to C.

In our fingerprinting method, the dispatch step might divide the original data into N chunks, N being the number of faulty processors, and send each chunk to only one processor. This is in contrast to TMR and STMR that send the whole data to all processors. In fingerprinting, the *voting* step would generate fingerprints on each chunk of data and would determine if it is calculated correctly. This again is in contrast to TMR and STMR, where data from N different processors must be transmitted and compared. The *repair* step is the same for both STMR and fingerprinting.

IV. Applications

We have implemented the parallel version of a number of applications and have developed fingerprinting and/or STMR verification processes for them. Each of the subsections below presents details on each application.

A. Matrix Multiplication (MM)

We use the Freivalds algorithm except that instead of generating a binary vector $\alpha\in\{0,1\}^n$, we create a vector $\alpha=(p_1,p_2...p_n)$, $p_i\in\Pi$, where Π is a pre-computed set of prime numbers (in our experiments $|\Pi|=17984$). Using prime numbers instead of binary will result in significantly better chances of catching errors.

We also use STMR with K being selected dynamically as follows: for problem size R (two $R\times R$ matrices to multiply), there are about $2\times R$ atomic operations to compute a cell, and the probability of a cell being faulty is $p_{cellFault}=1-(1-\rho)^c$, c being the number of atomic operations to be done for one cell. We chose the base partition size to be proportional to $1/p_{cellFault}$.

B. FFT on Real Input Vectors

We make use of the observation that in FFT operations performed on inputs that have only real components, the output is going to be symmetric. If the input to FFT is vector $\bar{x}=(x_1,x_2,...,x_k)$ and the output is $\bar{y}=FFT(\bar{x})$, then the following symmetry properties hold for all components except for $i=1$, $k/2+1$: $Re(y_i)=Re(y_{k-i+2})$, $Im(y_i)=-Im(y_{k-i+2})$. We use this property as the fingerprint, comparing elements of the output to see if the symmetrical ones match.

C. Bubble Sort (BS)

In the dispatch step of bubble sort implementation, we first down-sample the input by randomly selecting a small set from the input (regardless of the number of processors, 10% of the input size in our experiments) and use one of the N faulty processors to sort it. The sorted sample would give us a good idea about the distribution of numbers, which in turn would help us only concatenate the results from the faulty processors after the whole data is sorted. The values in the down-sampled sorted vector act as $N-1$ threshold values T_i ($i\in\mathbb{N}$, $i\leq N-1$) to decide how to distribute the original input among processors: we send numbers that are between T_{i-1} and T_i to the i^{th} faculty processor for sorting. If the down-sampled vector is statistically a good representation of the input number distribution, we will end up with a good load balancing among faulty processors.

The results from all faulty cores are gathered and fingerprinting is applied. We use three fingerprinting methods done by the robust processor: (1) check all elements from a faulty core to see if they are in the right order. If not, reject the result and send for repair. We call this strategy "total check". (2) check only 50% of the numbers to see if they are in the right order. This is called "50% check". (3) finally, the "smart check" methods checks the list from the end to see if the elements are in the right order. If elements $i+1..k$ out of the k elements are in the right order, we first check to see if elements $i+1..k$ are the largest of this batch, and then send only elements $1..i$ for repair.

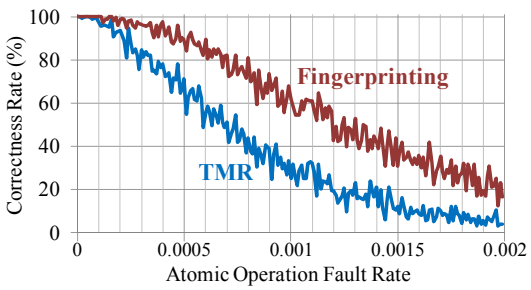
D. AES Encryption

We used a key (block) size of 16 to encrypt texts of size 16000 bytes. Partition sizes of STMR are selected statistically based on the following estimations (see Section IV, Matrix Multiplication): there are about 600 atomic operations to compute a cell. We chose the base partition size to be proportional to $1/p_{cellFault}$, e.g., for $\rho=10^{-5}$, we get $K=\sim 160$. Then, we chose fixed partition sizes around 160.

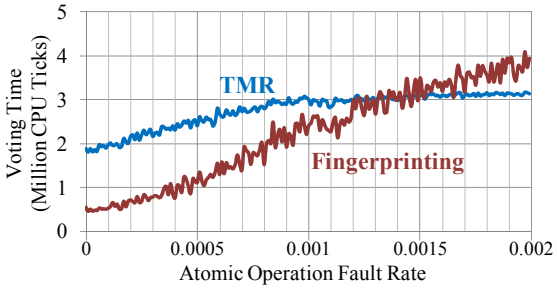
V. Experimental Results

A. Simulation Environment

We have developed a simulation environment that emulates parallel execution of the cores and keeps track of their runtimes. We artificially insert random errors in the output of computations of the applications that we implement in this environment based on the atomic fault rate ρ. Runtimes of faulty and robust cores are measured using clock ticks and milliseconds (operating system calls) and scaled based on their corresponding speedups. Since the size of partitions in STMR affects the results, several experiments were performed with different partition sizes for

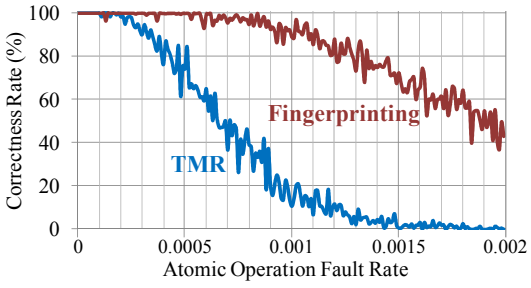


(a) average output correctness rate

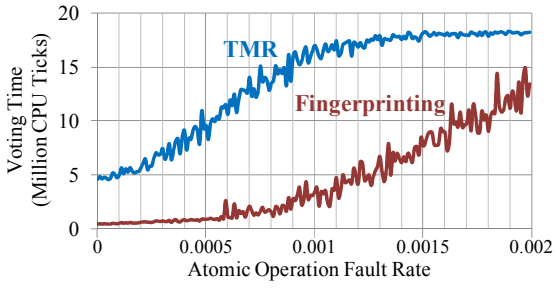


(b) average voting time

Fig. 2. FFT size 1024, 3 faulty cores. Each point shows the average of 25 Monte-Carlo runs.

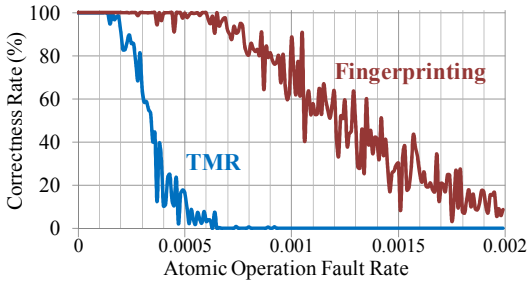


(a) average output correctness rate

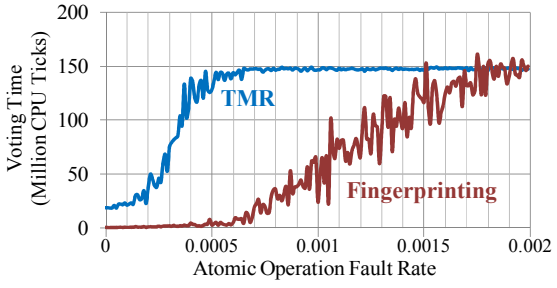


(b) average voting time

Fig. 3. FFT size 1024, 9 faulty cores. Each point shows the average of 14 Monte-Carlo runs.



(a) average output correctness rate



(b) average voting time

Fig. 4. FFT size 2048, 19 faulty cores. Each point shows the average of 5 Monte-Carlo runs.

MM Dynamically and AES Statically. The chosen partition sizes were around the estimations provided in Section IV. For example the term $PSR = 0.5$ in Fig. 6 and Fig. 7 means that the partition size for a given fault rate is half the size estimated by our calculations. Since our purpose is to compare the effectiveness of our algorithms to that of raw TMR, we have skipped the re-computation in the repair step for MM, FFT and AES.

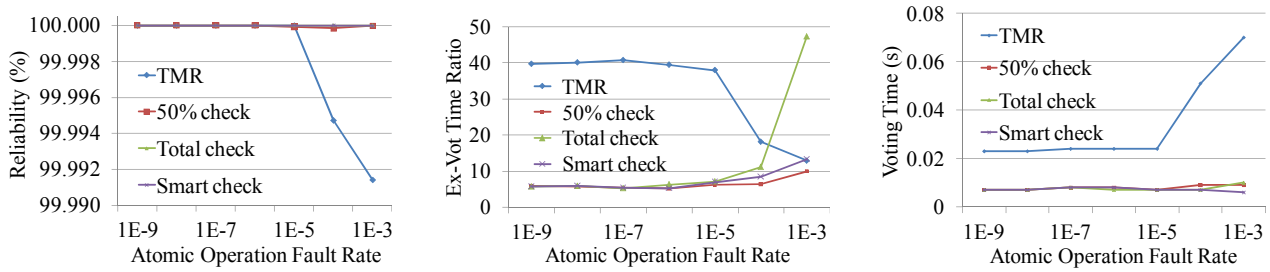
B. Fingerprinting vs. TMR

We performed experiments in which we ran Monte-Carlo simulations with high resolution of the ρ value and compared TMR with our fingerprinting methods.

Fig. 2 to Fig. 4 show the results of these experiments. The x-axis in these graphs is the atomic fault rate ρ. Part (a) of the figures shows average output correctness results (the y-axis is the percentage of the elements of the output vector that are correct), and part (b) of the figures shows the voting / fingerprinting time in terms of CPU ticks. To show the detailed behavior of the algorithms, we have used many points on the x-axis (200 values for the fault rate between 0 and 0.002) and have used CPU ticks for voting time. This is in contrast to the rest of the graphs in the paper in which we use only a handful of values of the atomic fault rate and use milliseconds for reporting runtimes. Input size is 1024 for Fig. 2 and Fig. 3 and 2048 for Fig. 4. Three, 9 and 19 faulty cores are used with one robust processor in Fig. 2, Fig. 3 and Fig. 4 respectively[1].

The results show that our fingerprinting method clearly outperforms TMR. Both methods produce more reliable outputs when the number of cores increases, but fingerprinting is by far more effective in utilizing the processors for better reliability, as is evident from the growing gap

[1]Fewer runs per ρ value in Fig. 3 and Fig. 4 were used to cut on simulation times. This has resulted in more fluctuations in the output, though.



(a) output reliability (b) ratio between voting and execution times (c) voting time

Fig. 5. Bubble Sort (BS) size 10000, 7 faulty cores that are 7x faster than the robust core. Each point shows the average of 11 Monte-Carlo runs.

between the two graphs as the number of processors increases. Note that the fingerprinting method is able to keep the output correctness rate close to 100% even for relatively high values of ρ when the number of processors increase from 3 to 9 (see Fig. 3(a)). But as the input size grows, the number of levels in the "butterfly" operations of FFT increases, which means an error at an early stage would propagate to more output entries and thus output correctness rates drop faster in the fingerprinting graph of Fig. 4(a) compared to Fig. 3(a).

Fig. 5 shows the results of the comparison between TMR and fingerprinting on the bubble sort application. Seven faulty cores are used. The dispatch strategy described in Section IV is employed. As can be seen, the fingerprinting method results in much better reliability and runtime.

C. STMR vs. TMR

In this subsection, we compare our proposed STMR to the traditional TMR. The results of the MM application are presented in Fig. 6 and Fig. 7. Fig. 6 shows the results when three faulty processors are used and Fig. 7 shows the results for seven faulty processors. It can be seen that the voting time is much smaller in STMR compared to TMR for small and medium atomic fault rates. It can also be seen that the number of successful signature checks largely depends on the partition size.

Fig. 6(a) and Fig. 7(a) show that for low fault rates, STMR voting times are well below the TMR line. As mentioned before, partition size is selected dynamically and reduces when the atomic fault rate increases to keep the voting time as small as possible[2].

AES results are very similar to MM results and a sample is presented in Fig. 8.

D. Fingerprinting vs. STMR

We can compare the performance of fingerprinting to STMR in Matrix Multiplication in Fig. 6(a) and Fig. 7(a). It can be seen that when 3 faulty processors are used, fingerprinting cannot adapt to the high fault rates and the voting time increases rapidly. On the other hand, STMR with proper partition size shows more stability against the increment in the fault rate. When 7 faulty processors are used, fingerprinting outperforms STMR because it is more efficient in utilizing the processors.

[2]As the atomic fault rates increase, the algorithm changes its U-curve behavior shown in Fig. 9 and jumps from a curve with low output fault rate to a curve with a higher output fault rate associated with the new atomic fault rate. The U-curve is explained in the appendix.

VI. Conclusion and Future Work

In this paper, we proposed a new programming style in which parallelism, reliability and fault tolerance are explicitly specified in the code. Our future work includes more general architectures in which levels of faulty processors can be connected together, and more accurate models of the synchronization costs and communication faults.

APPENDIX: Modeling the Cost of STMR

As we mentioned in Section III, there is a tradeoff between voting time and repair time (N, R and K defined in Section III). The Total cost of STMR is as follows:

$$Cost_{STMR} = Cost_{Sig} + Cost_{Voting} + Cost_{Repair}$$

Assuming that $N-\alpha$ ($\alpha \in \mathbb{R}$, $0 \le \alpha \le N-1$) is proportional to the mean number of comparisons throughout the TMR process, we have:

$$Cost_{Sig}(R) = R - \lceil R/K \rceil$$
$$Cost_{TMR}(R,N) = R(2N-2\alpha+1)(N-1)/2$$

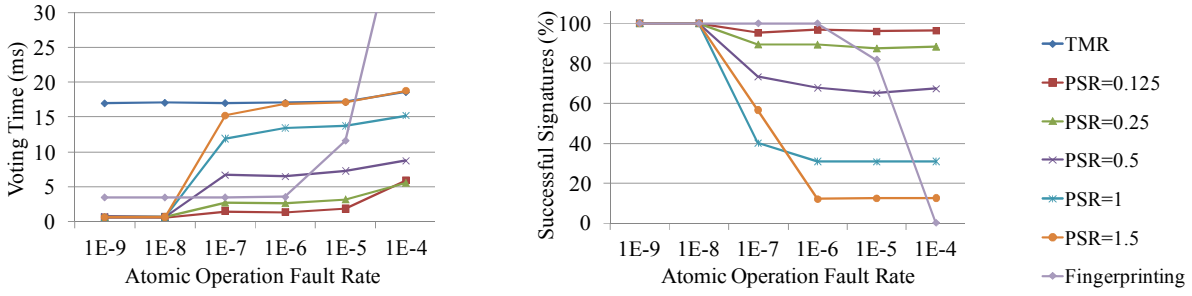
Note that the cost of memory access is ignored. We define "Leakage" to be the number of signatures that fail voting and are sent back for repair. The lower and upper bounds of Leakage for a given output fault rate FR are as follows:

$$Leakage = L = \begin{cases} min: \lceil FR \times R/K \rceil \\ max: min\{FR \times R, \lceil R/K \rceil\} \end{cases}$$

Therefore, the total cost of STMR is:

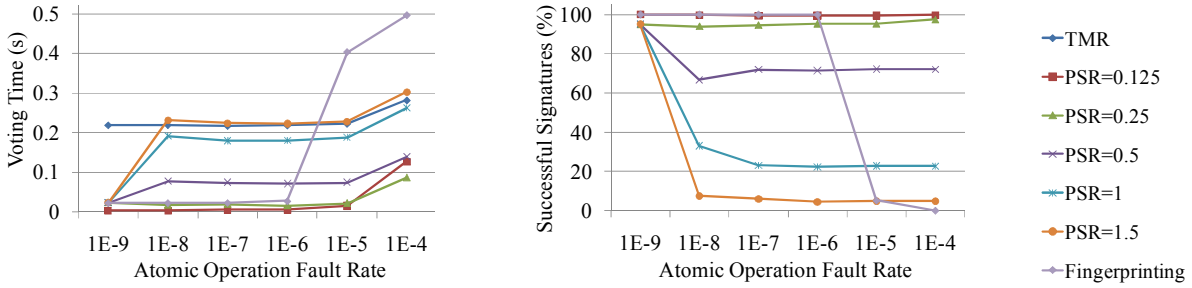
$$Cost_{STMR}(R,N) = Cost_{Sig}(R) + Cost_{TMR}(\lceil R/K \rceil, N) + Cost_{TMR}(L \times K, N)$$

According to the formula, the growth of cost in TMR on signatures plays against that of repair as long as there are signatures extracted in voting that are not sent for repair. This leads to a curve similar to "U" when the cost is plotted for different partition sizes. A typical U-Curve for a specific error distribution ($\alpha = 2$), $N = 5$ and $R = 10^6$ is shown in Fig. 9, in which different curves correspond to different output fault rates.



(a) voting time (b) percentage of successful STMR signatures

Fig. 6. Matrix Multiplication (MM), 3 faulty cores that are 7x faster than the robust core. Problem size 200x200.



(a) voting time (b) percentage of successful STMR signatures

Fig. 7. Matrix Multiplication (MM), 7 faulty cores that are 7x faster than the robust core. Problem size 500x500.

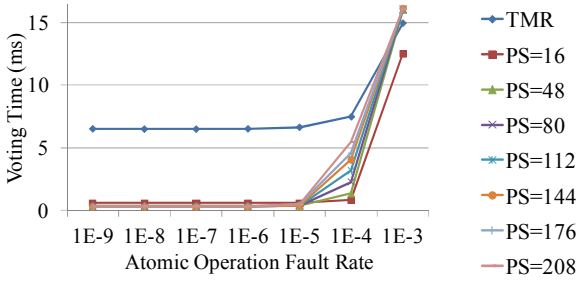


Fig. 8. AES voting time, 3 faulty cores. Problem size 16000.

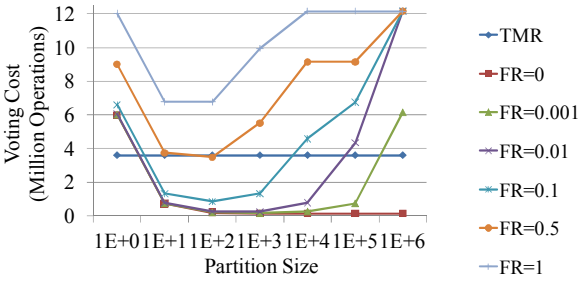


Fig. 9. Voting cost as a function of ρ and K.

References

[1] International Technology Roadmap for Semiconductors (ITRS), http://public.itrs.net/.

[2] M. Butts, A. DeHon, and S.C. Goldstein, "Molecular Electronics: Devices, Systems and Tools for Gigagate, Gigabit Chips," *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 433-440, November 2002, San Jose, CA.

[3] C. Visweswariah, "Death, taxes and failing chips," *Proc. 2003 Design Automation Conference (DAC)*, pp. 343-347, June 2003, Anaheim, CA.

[4] C. Visweswarih, K. Ravindran, K. Kalafal, S. G. Walker, and S. Narayan, "First-order incremental block-based statistical timing analysis," *Proc. 2004 Design Automation Conference (DAC)*, pp. 331-336, June 2004, San Diego, CA.

[5] H. Chang and S. S. Sapatnekar, "Statistical timing analysis considering spatial correlations using a single PERT-like traversal," *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 621-625, November 2004, San Jose, CA.

[6] J. Xiong, V. Zolotov, C. Visweswariah, and N. Venkateswaran, "Criticality computation in parameterized statistical timing," *Proc. 2006 TAU (ACM/IEEE workshop on timing issues in the specification and synthesis of digital systems)*, pp. 119-124, February 2006, San Jose, CA.

[7] X. Li, J. Le, M. Celik, and L. T. Pileggi, "Defining statistical sensitivity for timing optimization of logic circuits with large-scale process and environmental variations," *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 844-851, November 2005, San Jose, CA.

[8] K. Chopra, S. Shah, A. Srivastava, D. Blaauw, and D. Sylvester, "Parametric yield maximization using gate sizing based on efficient statistical power and delay gradient computation," *IEEE International Conference on Computer-Aided Design (ICCAD)*, pp. 1020-1025, November 2005, San Jose, CA.

[9] S. Nassif, "Design for variability in DSM technologies," *IEEE International Symposium on Quality Electronic Design (ISQED)*, pp. 451-454, March 2000, San Jose, CA.

[10] M. B. Tahoori and S. Mitra, "Fault Detection and Diagnosis Techniques for Molecular Computing," *Proc. 2004 Nanotech*, Vol. 3, pp. 57-60, March 2004, Boston, MA.

[11] M. B. Tahoori, "Defects, Yield, and Design in Sublithographic Nanoelectronics," *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pp. 3-11, October 2005, Monterey, CA.

[12] M. B. Tahoori, J. Huang, M. Momenzadeh, and F. Lombardi, "Defect and Fault Characterization in Quantum Cellular Automata," *Proc. 2004 Nanotech*, Vol. 3, pp. 190-193, March 2004, Boston, MA.

[13] P. Maidee and K. Bazargan, "Defect-tolerant FPGA Architecture Exploration," *16th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1-6, August 2006, Madrid, Spain.

[14] J. Hromkovic, "Design and Analysis of Randomized Algorithms: Introduction to Design Paradigms," *Springer*, 2005.

[15] R. Freivalds, "Fast probabilistic algorithms," *Proc. 8th Symposium on Mathematical Foundations of Computer Science*, pp. 57-69, 1979.