Architectural-Level Fault Tolerant Computation in Nanoelectronic Processors *

Wenjing Rao
UC San Diego
CSE Department
wrao@cs.ucsd.edu

Alex Orailoglu
UC San Diego
CSE Department
alex@cs.ucsd.edu

Ramesh Karri
Polytechnic University
ECE Department
ramesh@india.poly.edu

ABSTRACT

Nanoelectronic devices are expected to have extremely high and variable fault rates; thus future processor architectures based on these unreliable devices need to be built with fault tolerance embedded so as to satisfy the fundamental requirement of computational correctness. In this paper an architectural-level computation model is proposed for fault tolerant computations in nanoelectronic processors.

The proposed scheme is capable of guaranteeing the correctness of each instruction through exploitation of both hardware and time redundancy, even under high and variable fault rates. Each instruction is confirmed by multiple computation instances. Through a speculative execution based on unconfirmed results, the proposed scheme eliminates the severe performance deterioration typically caused by time redundancy approaches on data dependent instructions. To avoid the exponential growth of resource allocation introduced by the hardware redundancy approaches on the speculations, a hardware allocation framework is developed in the proposed scheme to control the growth of hardware resources while preserving the low latency achieved through the speculative executions.

We set up an experimental framework to validate the effectiveness of the proposed scheme as well as to investigate multiple tradeoff points within the proposed approach. Experimental data further confirm that the proposed approach achieves the goal of providing fault tolerance in the pipelined nanoelectronic processors, while at the same time providing high system performance and efficient utilization of hardware resources.

1. INTRODUCTION

As CMOS is approaching its physical limit, nano-scale devices such as Resonant Tunneling Diodes [1], Quantum-dot Cellular Arrays [2] and molecular electronics [3] have been proposed due to their drastically improved operating speeds, low power consumption and device densities reaching 10^{12} device / cm^2 [4, 5].

However, the defect rates in these emerging nano devices are projected to be in the order of $10^{-3} - 10^{-1}$, in comparison with defect rates of $10^{-9} - 10^{-7}$ in CMOS technology [5]; additionally, a high occurrence of time varying transient faults is expected at run-time [5, 6, 7]. Consequently, defect and fault tolerance has assumed paramount importance as a design objective in the emerging nanoelectronic environment.

Related research in defect tolerance in the nanoelectronic systems includes system level approaches [8] and logic level solutions [9, 10]. These approaches reconfigure the redundant hardware to bypass the permanent faulty units. To deal with transient faults and provide fault tolerance capability, the N Module Redundancy and NAND multiplexing can be applied at the logic gate level [11, 12]. These simple hardware redundancy based schemes at the logic and lower levels require an immense number of redundancies to tolerate the large failure rates in the emerging nanotechnologies [13]. On the other hand, research approaches in architectural-level fault tolerance schemes for processors based on CMOS technology [14, 15, 16, 17] mainly deal with a low and relatively fixed fault rate and are not applicable to the nanoelectronic environment.

Essentially, computational models for computing system architectures based on nanoelectronic devices should satisfy two core requirements. First, correctness of computations is a fundamental requirement. The overall system should operate reliably even though the underlying nanotechnology is unreliable. A second requirement is to implement a high-performance system. The large number of computation units should be used to dramatically speed up system performance. In doing so, several unique challenges need to be addressed including (i) How to translate the speed up afforded by nanoelectronic devices into high performance at the system level and (ii) How to organize the abundant computational resources to trade off fault tolerance against system performance in the presence of high and time varying failure rates.

In this paper, we propose a fault tolerance computational model for the nanoelectronic processors. Essentially, the correctness of every instruction is confirmed by multiple execution instances through hardware redundancy complemented by time redundancy approach. To achieve system performance, multiple unconfirmed computation branches can proceed in parallel in a speculative manner. Hardware resource growth in the speculative computations is controlled so that performance boost does not occur at hardware expense. An instance of the proposed computation model can be found in [18].

We set up an experimental framework to validate the effectiveness of the proposed approach and investigate the variations of the algorithm under different parameters. Experimental data confirm that, both in terms of hardware and in terms of latency aspects, the proposed computation model can provide fault tolerance for pipelined instructions even under a high and variable fault rate and can achieve high system performance with low hardware overhead. The experimental results further show that, for different fault rate ranges, a solution quite close to the optimal can be achieved through the proper selection of parameters in the proposed computation model.

The paper is organized as follows. We first provide overall motivation in section 2. A description of the proposed fault tolerance computation model follows in section 3, with the hardware allocation algorithm for a pipelined architecture provided in subsection 3.2. Experimental setup, results and analysis are provided in section 4. A brief set of conclusions is offered in section 5.

2. MOTIVATION

Triple modular redundancy (TMR) and its generalized form, N modular redundancy (NMR), have been some of the most commonly applied approaches for fault tolerance. To apply this straightforward strategy, an instruction can be computed by N distinct units in parallel

*The work of the first two authors is supported in part by NSF Grant 0082325.

and the result confirmed by a majority vote. This hardware redundancy based strategy is supported by the emerging nanotechnologies due to the abundant hardware resources.

A careful analysis reveals however that this approach is practical only if the fault rates are low or steady enough to yield confirmed results, since the redundancy is predefined. When the fault rate is high, the predetermined number of computation units might generate distinct results and lead to an unconfirmable computation. On the other hand, setting the redundant computation unit number to be very large consumes unnecessary hardware overhead if the fault rate is comparatively low. Therefore, the inflexibility of the NMR fault tolerance strategy makes it extremely hard to match the predefined amount of redundancy with the high and variable fault rates in effect in the nanoelectronic environment, thus limiting its applicability severely.

To deal with the high and varying fault rates, time redundancy can be utilized in a manner complementary to hardware redundancy for flexibility. With time redundancy, an instruction is always confirmable despite the possibly high fault rate, since it can always allocate new computation units at the next cycle when the current results all disagree.

However, applying a pure time redundancy approach typically costs prolonged latency in the confirmation process. The latency introduced by time redundancy becomes a severe problem when data dependency exists among instructions, especially in a pipelined environment. It might take an unpredictable number of cycles, as determined by the time varying fault rate, before an instruction can be confirmed. Therefore, a successor instruction that relies on the unconfirmed result of a current instruction has to be delayed, resulting in a domino effect on subsequent instructions and a tremendous number of stalls in an instruction pipeline. Therefore, overall system performance can be severely compromised by the time redundancy approach.

To solve this problem, suppose an instruction B takes the result of instruction A as an input, while A takes an exceedingly long time to confirm. It can be observed that B might start executing without delay by taking the multiple unconfirmed results of A speculatively. Multiple *speculative branches* in B's computation can thus be formed. As the result of A is confirmed, the correct branches of B are retained and the incorrect speculative branches are pruned. The results in agreement within a correct branch of B can further confirm B itself.

It is worth noticing that the formation of multiple speculative branches necessitates at least twice the amount of the hardware resources to compute all the next level branches in parallel, since the initial NMR needs to be applied in each speculative branch in order to locally confirm the instruction itself.[1] Therefore, it represents a mechanism with low latency, yet high hardware overhead. Therefore, while speculation can speed up instruction execution in the presence of data dependencies, speculative branches can grow exponentially and exhaust rapidly even the abundant hardware available in a nanoelectronic environment.

Two extremal positions in terms of hardware vs. latency tradeoff can thus be envisioned.

- In a *no speculation* approach, if the input data of an instruction depends on another instruction and is not yet confirmed, the instruction is delayed and waits for its confirmed input, thus necessitating no extra computation units for speculations. Therefore, it represents a mechanism with low hardware, yet high latency overhead.
- In a *full speculation* approach, an instruction can start execution by generating multiple speculative branches for every unconfirmed input, thus representing a mechanism with high hardware and low latency overhead.

To avoid the severe latency problem in the *no speculation* approach and the exponential growth of hardware allocation in the *full speculation* approach, a hardware allocation framework is proposed to achieve both frugal hardware resource allocation and short overall latency. The key idea is to generate speculative branches and allocate hardware resources on an as-necessary basis. In other words, extra hardware is only allocated when the instruction is definitely not confirmable due to high occurrence of faults.

[1] We assume each speculative branch always allocates the minimum number, i.e., two redundant computations for local confirmation purposes.

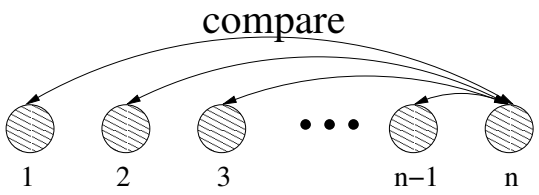

Figure 1: Comparison performed when the nth result is returned

3. NANO PROCESSOR COMPUTATION MODEL

The proposed computation model for a nanoelectronic processor essentially consists of two parts: (i) the fault tolerant scheme that utilizes hardware and time redundancy to guarantee computation correctness; (ii) the hardware allocation algorithm that performs a tradeoff between hardware resources and system performance when dealing with speculations in dependent instructions.

3.1 Fault Tolerance Computation

In order to guarantee the correctness of the computations in the nanoelectronic environment where fault rates are high and variable, an instruction should be confirmed by at least two results in agreement.[2] The fault tolerance computation is composed of an initial NMR hardware redundancy approach followed by the time redundancy approach, which continues invoking new computation instances until any two results agree.

Intuitively, a larger quantity of hardware redundancy allocated at the initial cycle enables faster confirmation when the fault rate is high. In this section, we provide an initial exposition of the proposed scheme based on the minimum case of 2 initial C-units. In order to investigate the tradeoffs between hardware and latency for different N in the initial NMR hardware redundancy approach, we develop a series of experiments with the results discussed in section 4.

In the instruction processing system, we make the following assumptions. First, for the purposes of fault tolerance, the system needs a pool of computation units (denoted as *C-unit*) to carry out the redundant execution instances for each instruction. Secondly, to manage the hardware and time redundancy to achieve fault tolerance, a control unit (denoted as *voter*) is dedicated for each instruction and is responsible for returning a confirmed result.

A voter is allocated when an instruction is issued. Meanwhile, the issuing machine continues to process the next instruction in a pipelined manner with no delay incurred. Initially, the C-units are allocated by the voter to execute the instruction in parallel. The C-units then return the results to the voter and are released. The voter stores and compares the two initial results. If the two results agree, the instruction is confirmed and the voter is released. Otherwise, the voter incrementally applies time redundancy by allocating one C-unit each time until two of the results agree to confirm the instruction.

The number of comparisons performed inside a voter equals the number of existing results stored in the voter. Since the previous stored results in a voter should all be distinct, the only possible agreement is between one of the previously stored results and the newly returned result. Figure 1 depicts the comparisons needed to be performed upon the nth result being available.

[2] It is presumed that faults in computation units exhibit themselves in distinct ways; further insurance against letting faulty computations slip through can be attained by increasing the threshold of agreement.

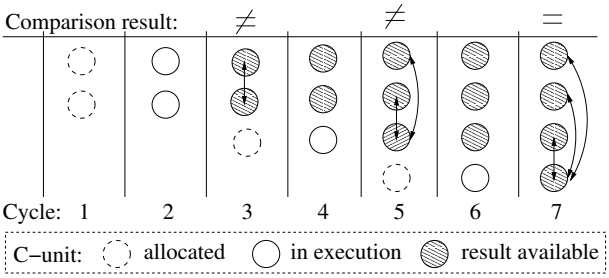

Figure 2: An example of the instruction confirmation process

Due to the time redundancy approach involved in the proposed fault tolerance scheme, the confirmation of an instruction might suffer unpredictable latency. Therefore, the only reasonable approach to guarantee system performance is to divide the confirmation process into multiple cycles and utilize a pipeline architecture to process consecutive instructions. We assume the following pipeline stages for the instruction process:

- The instruction decode and initial allocation of C-units.
- The execution of the instruction by a C-unit.
- The comparison of results and allocation of new C-units (if needed).

Figure 2 shows an example of an instruction being confirmed in seven cycles with four C-units allocated in total. In cycle 1 the voter initially allocates two C-units and the results are available in the third cycle. The two C-units are released at this point and the results are stored in the voter. Since the results disagree, one more C-unit is allocated. After one cycle of execution, the new result is compared with the stored two results. Since no agreement is attained through the comparisons, another C-unit is allocated in cycle 5 and the instruction is eventually confirmed at cycle 7 when one of the three previous stored results agrees with the newly returned result.

3.2 Dynamic hardware allocation algorithm

In order to improve system performance, in the proposed architecture, an instruction that depends on a yet unconfirmed result from another instruction (denoted respectively as the *child instruction* and *parent instruction*) speculatively uses the result without the delay of confirmation. As the comparison result is obtained by the voter for the parent instruction, it is used to either confirm or prune the speculative branches of the child instructions.

Based on the status of a branch and the comparison within the branch, any result in a speculative branch can fall into one of four categories:

- **Correct branch:** both the branch and the result are confirmed to be correct; thus the instruction can be confirmed with this result.
- **Wrong branch:** either the branch or the result itself is confirmed to be wrong; thus all the children branches of this result should be pruned.
- **Global:** the speculative branch is confirmed to be correct but the correctness of the result has not been confirmed.
- **Locally confirmed:** the result is confirmed to be correct within the branch but the correctness of the branch has yet not been confirmed.

In order to provide local confirmation capability within each branch, the *full speculation* approach always allocates two C-units for every branch, thus resulting in an exponential growth of required hardware resources for a sequence of child instructions. Basically, the strategy assumes that every unconfirmed result from the parent level will be distinct, by utilizing an extremely pessimistic overestimation of the fault rate.

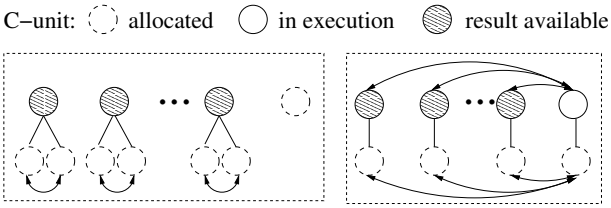

Figure 3: Initial allocation of C-unit in a child instruction

Essentially, an ideal hardware management algorithm for the child instructions needs to achieve the goals of frugal hardware resource allocation, quick confirmation with low latency and fault tolerance for the computation. Therefore, the following aspects should be considered:

- The initial C-unit allocation for a child instruction should try to preserve the confirmation possibility of the instruction with minimum hardware resources, instead of simply assigning two C-units for each speculative result.
- Hardware allocation should be frugal for a child instruction as long as there still remain possibilities for confirmation.
- Parent instructions should be provided with a higher priority for obtaining hardware resources since child instructions are executed on a speculative basis and hardware resources can be easily wasted on incorrect branches.
- When a child instruction becomes impossible to confirm, new C-units should be allocated based on the updated speculation information.

Essentially, when the fault rate is low, the initial speculative branches suffice for the child instruction to quickly confirm, without consuming large amounts of hardware resources. On the other hand, when the fault rate is high, the speculation branches should be controlled to a limited number to avoid an exponential growth of hardware requirement. Hardware resources need to be utilized, in this case, mainly on the root level of the speculation tree, thus guaranteeing the early confirmation of the parent instructions.

We explore the hardware allocation algorithm in depth in the following three subsections.

3.2.1 Initial C-unit allocation of child instructions

In order to confirm an instruction when no fault occurs with minimum hardware resources in the shortest time, every instruction should be initially allocated with two C-units. However, for a child instruction based on an unconfirmed result, the full speculation approach allocates two C-units for each result, which is unnecessary. This is because two of the unconfirmed results in the parent instruction eventually turn out to agree, while the initial allocation of the child instruction presumes every unconfirmed result to be distinct.

To control the growth in hardware resources, when a child instruction is issued, information from the previous comparison that has occurred in the parent instruction should be utilized. In other words, for an unconfirmed result, although its correctness may not be ascertainable, the information of whether it *might* be in agreement with another unconfirmed result or is definitively distinct may be available. For every unconfirmed result in the parent instruction, the initial C-unit allocation for the child instruction is described below:

- if the result is marked as a *wrong branch*, no C-unit allocation.
- if the result is known to be distinct, allocate two C-units.
- if the result might display agreement with another unconfirmed result, allocate only one C-unit.

Figure 3 shows the specific initial C-unit allocation cases for a child instruction. In the case shown on the left side, all the results in the

parent instruction are known to be distinct since the comparison information is available. Meanwhile, a new C-unit is being allocated in the parent instruction. For the child instruction, two C-units are initialized for each distinct result, forming multiple speculation branches that can be locally confirmed. There is no need to allocate any C-units in the child instruction to take the result of the newly issued C-unit in the parent instruction, since its result will not be available until two cycles later.

However, in the case shown on the right side of figure 3, all the available results in the parent instruction are to be compared with the last C-unit, the result of which will be available in the next cycle. Therefore, every result has a possibility to agree with the new C-unit in the parent instruction. In this situation, the child instruction allocates one C-unit for each result, and builds the same comparison relationship as the parent instruction. A C-unit is also allocated in the child instruction for the executing C-unit in the parent instruction, since its result will be available in the next cycle and can be passed to the child C-unit for execution.

3.2.2 Information propagation

The information obtained from a parent instruction comparison needs to be propagated to all the child instructions, so as to direct the branch pruning/confirmation and hardware allocation in the child instruction.

When a pair of unconfirmed results in a parent instruction is compared, if a child instruction had initially allocated one C-unit for each of the results, then the two C-units in the child instruction are scheduled to be compared with each other since their parent C-units might agree, just as is shown in the right hand part of figure 3. Therefore, when the result C-unit pair in the parent instruction is in disagreement, the information should be propagated to cancel the comparison between the two child C-units, since they are presumed to disagree due to their distinct data inputs.

If the result pair in the parent instruction is in agreement and the results are marked as *global*,[3] then the parent instruction can be confirmed with the agreeing pair of results. Otherwise, the pair of results becomes *locally confirmed*.

When a parent instruction is confirmed, information is passed to preserve the correct and prune the wrong speculation branches. The children of the results with the *correct branch* are marked as *global*, indicating that the data inherited from the parent instruction is correct. If the results of the children are already *locally confirmed*, then they become a *correct branch* and the child instruction can be confirmed too. The results of the parent instruction marked as a *wrong branch* will propagate the information to all the descendants and prune the speculation branches.

3.2.3 C-unit update

To control the growth of hardware resources, in a child instruction, other than the initial C-unit allocation, a voter only allocates new C-units when the instruction is known to be unconfirmable at the current stage. Essentially, there are two distinct situations where new C-units need to be allocated:

- If all the results in the child instruction are marked as a *wrong branch*, then confirmation of the instruction is no longer possible. The new C-units can only be allocated from the parent instruction, taking the results which are not marked as a *wrong branch*. This process is identical to the initial C-unit allocation in the child instructions.
- If there exist some unconfirmed results in the child instruction that are not *wrong branches*, but all known to be distinct, then the instruction is not possible to confirm since it lacks the local confirmation capability. Therefore, new C-units can be allocated by duplicating the computation of the unconfirmed results.

[3]If an instruction does not depend on the results of any unconfirmed instructions, i.e., it has no parent instruction, then all its unconfirmed results are marked as *global*.

4. SIMULATION RESULTS

We perform two sets of simulation to justify the efficiency and examine the possible variances of the proposed computation. First, the proposed fault tolerance computation model is compared with the *no speculation* and *full speculation* cases. Both hardware requirement and latency are compared for the three strategies. Through this set of experiments, we examine the tradeoff between latency and hardware by comparing the proposed scheme with the other two strategies. Secondly, we examine the proposed scheme with various replication quantities, N, for the initial NMR hardware redundancy approach. The results show the various tradeoff points existing within the proposed strategy and their corresponding applicability in different fault rates.

The simulation is implemented with a C++ program and is performed on a Linux machine. To model the typical system behavior in dispatching instructions, a sequence of instructions with possible data dependencies among them needs to be issued in order, i.e., the first one can be issued at the first cycle, while the ith instruction can be issued no earlier than the ith cycle. The experiment is based on a sequence of 50 instructions with randomly set dependencies, where each instruction can have a data dependency on any of the preceding instructions (as well as possibly no dependency). In the experiment, the results are obtained by taking the average number of 50 repeated experiments, with distinct dependency settings among the instructions.

4.1 Comparison with no/full speculations

To examine the tradeoff between hardware consumption and latency in the fault tolerance computation models, we set up an experimental framework to compare the proposed scheme with the two extremes in hardware and latency. Fault rates for the C-units are set to range from 0.01 to 0.30.

The *no speculation* approach issues a child instruction only after the confirmation of its parent instruction, i.e., no speculation of child instructions is ever performed. On the other hand, the computation model named *full speculation* extends every speculation branch and assigns two C-units for each in order to confirm the instruction as soon as possible.

4.1.1 Latency comparison

To measure the latency criterion, we examine the total number of cycles utilized to finish confirming all the fifty instructions. In the ideal case where no fault exists, an instruction needs three cycles to confirm, with the initial two C-units in agreement. Therefore, a sequence of instructions of length n would require at least $n+2$ cycles to complete since the nth instruction starts at cycle n at the earliest in a pipelined machine.

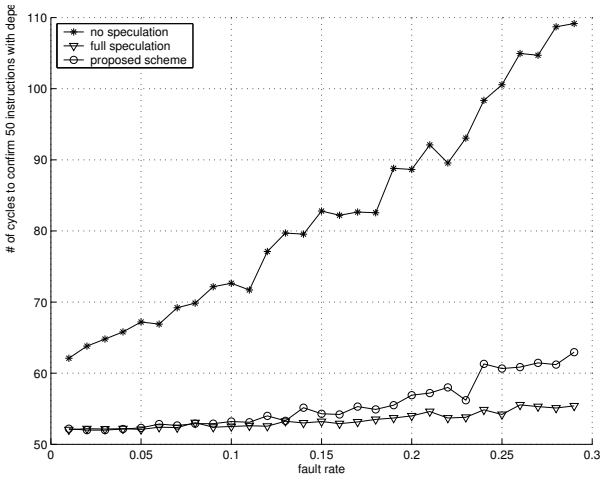

Figure 4: Latency comparison

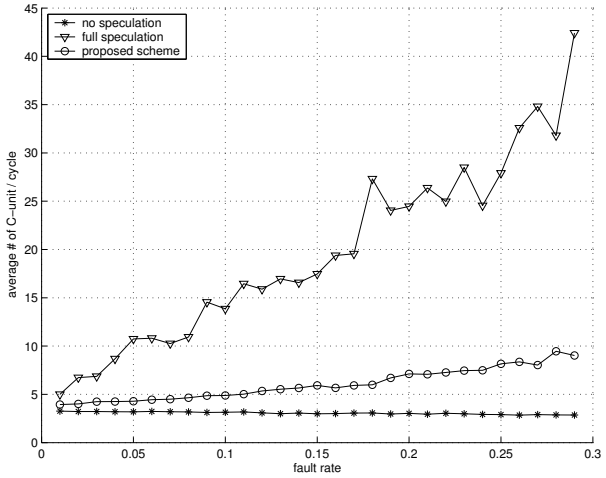

Figure 5: Avg C-unit HW comparison

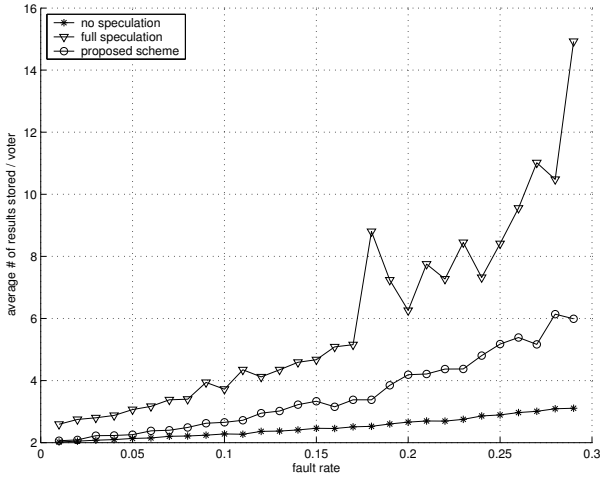

Figure 7: Avg Voter HW comparison

Figure 4 shows the latency comparison of the three computation models. The X axis shows the different fault rates and the Y axis indicates the number of cycles required to complete the confirmation of the 50 instructions. The starred line represents the *no speculation* approach, the line marked by triangles represents the *full speculation* approach while the line with circles indicates our proposed scheme.

It can be seen from the figure that the *no speculation* approach suffers from significant delays in comparison to the other two models. Since no speculation branch is ever formed, the data dependencies among the instruction sequences result in the delayed issuing of child instructions. The tremendous latency of the *no speculation* approach grows and makes the gap even larger as the fault rate increases.

We expect the *full speculation* approach to achieve the minimum latency since it utilizes unlimited hardware resources to provide the maximum redundancy for every speculation branch. The experimental data confirms this observation by showing that the line with triangles lies almost flat with nearly ideal performance even in the highest fault rates examined.

The proposed scheme, however, in terms of latency, exhibits behavior similar to that of the *full speculation* approach. When fault rates are below 0.15, the proposed scheme uses almost the same number of cycles as the best case of *full speculation*. For higher fault rates the proposed scheme only takes a few cycles longer than the minimum latency case of the *full speculation* for the 50 instructions altogether. Therefore, it is comparable to the *full speculation* case and outperforms the *no speculation* approach significantly.

Overall, simulation results confirm that the proposed scheme can achieve a near-optimum performance that is comparable to the best case of *full speculation*.

4.1.2 Hardware resource comparison

The hardware resource consumption in the fault tolerance computation models should be considered from two aspects. First, the number of C-units occupied at each cycle shows the amount of computational unit hardware requirement in the system. Second, since all the unconfirmed results need to be stored in the voter, the number of unconfirmed results during the computation depicts the storage requirement for each voter, which constitutes another portion of the hardware resource requirement.

Figure 5 shows the average number of C-units occupied for the three computation models. The X axis shows the fault rates while the Y axis shows the average number. Figure 6 shows the maximum number of C-units occupied for a cycle during the process of performing the 50 instructions, which indicates the "peak" of hardware resource consumption.

It can be seen from both figures that the three approaches perform consistently. The figures show that the *full speculation* approach exhibits erratic growth in C-unit allocation. The *no speculation* approach

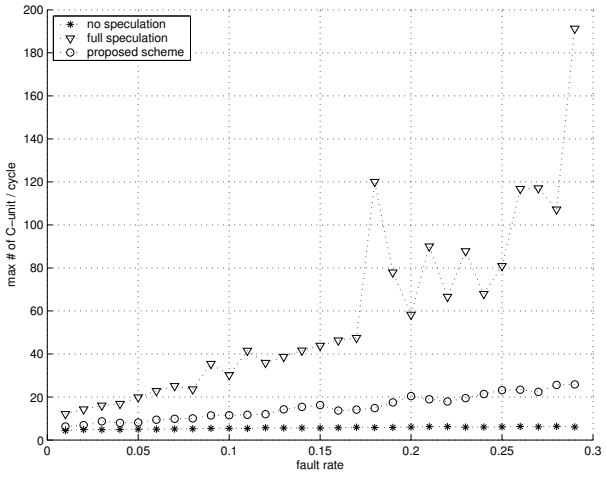

Figure 6: Peak C-unit HW comparison

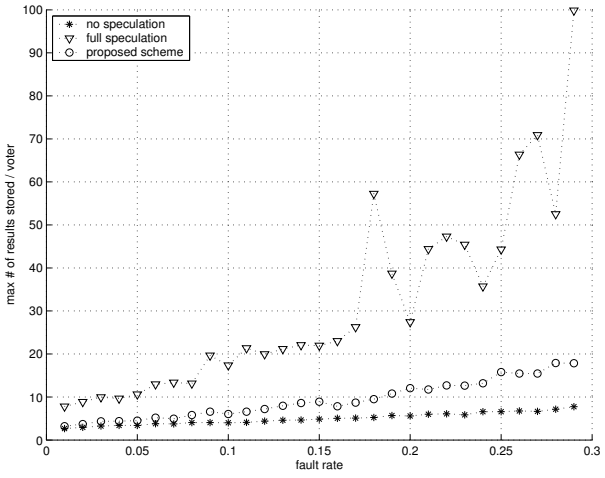

Figure 8: Peak Voter HW comparison

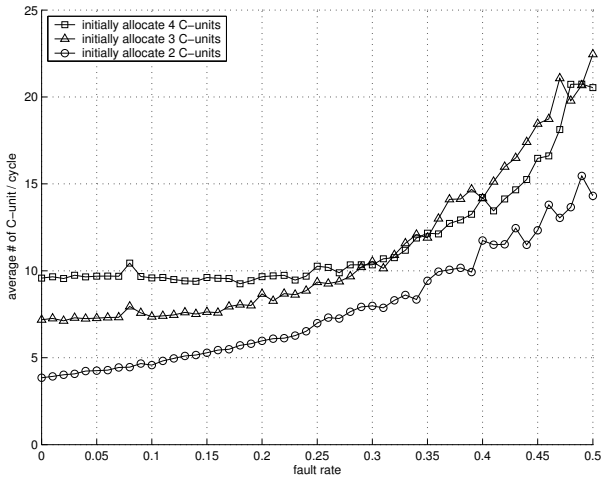

Figure 9: Avg C-unit HW comparison for various initial NMR #

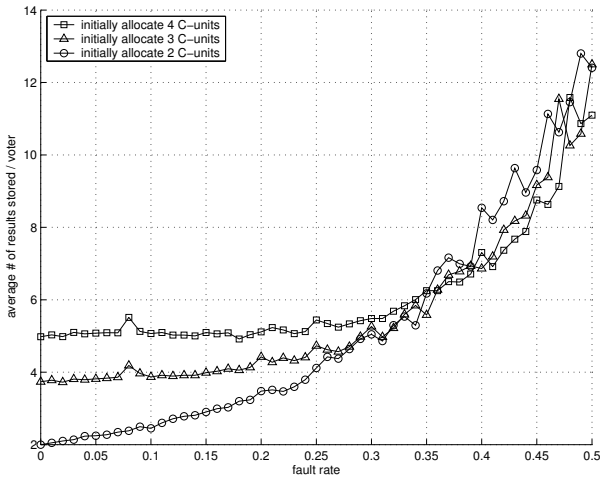

Figure 11: Avg Voter HW comparison for various initial NMR #

is always the steady lowest line since it never requires more than two C-units in each cycle. The proposed scheme avoids the exponential growth in allocating C-units with the dynamic control algorithm and exhibits a low increase as the fault rate is increased.

Figures 7 and 8 depict the hardware requirement in terms of voter storage. Figure 7 shows the average number of results stored in a voter for various fault rates, while figure 8 shows the maximum number of results that would ever need to be stored in a voter, i.e., the peak value.

Similar to the hardware consumption in C-units, the *full speculation* approach consumes an impractically high amount of resources, especially when the fault rate is high. The proposed scheme, on the other hand, shows a steady but quite flat growth and is only slightly higher than the *no speculation* approach, both in the average and in the maximum case.

Seen from the two aspects of hardware consumption, the proposed scheme represents an efficient hardware allocation strategy. This is represented by the simulation results showing that the proposed scheme significantly outperforms the *full speculation* approach and performs much closer to the *no speculation* approach in terms of both C-unit occupation and voter storage.

4.1.3 Overall analysis for latency/hardware tradeoff

Essentially, through the experimental results it can be observed that the proposed scheme shows significant results both in time and hardware aspects, thus achieving the goal of fault tolerance without sacrificing severely either hardware or latency. In comparing with the other two models, the proposed fault tolerance computation model can be seen to compete with the best aspects of the other two approaches, i.e., the short delay of the *full speculation* and the frugal hardware allocation in the *no speculation* without suffering any of their downsides. Thereby, the proposed scheme is shown to be a promising approach that overcomes the challenges and provides a possible fundamental computation model for nanoelectronic processors.

4.2 Further tradeoff investigation

The minimum initial NMR of C-unit allocation for the proposed scheme is two; however, the manner in which various initial C-unit settings influence the behavior of the proposed algorithm bears further scrutiny. In this experiment set, we compare the minimum case of 2 initial C-units with the cases of initially allocating 3 and 4 C-units. We expect the experimental results to provide us with knowledge of different tradeoff points within the proposed scheme.

With more initially allocated C-units, an instruction can be confirmed more quickly when the fault rate is high. On the other hand, when the fault rate is low, some initially allocated C-units are essentially redundant, thus consuming comparatively more hardware resources.

The simulation results confirm this syllogism. Figure 13 shows the results in terms of latency and figures 9 to 12 show the comparisons in

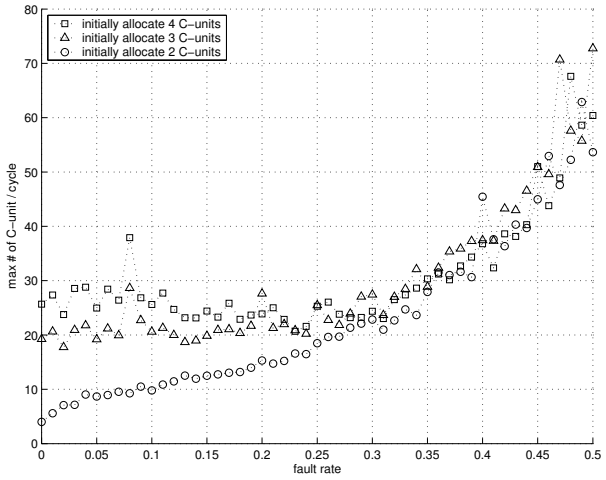

Figure 10: Peak C-unit HW comparison for various initial NMR #

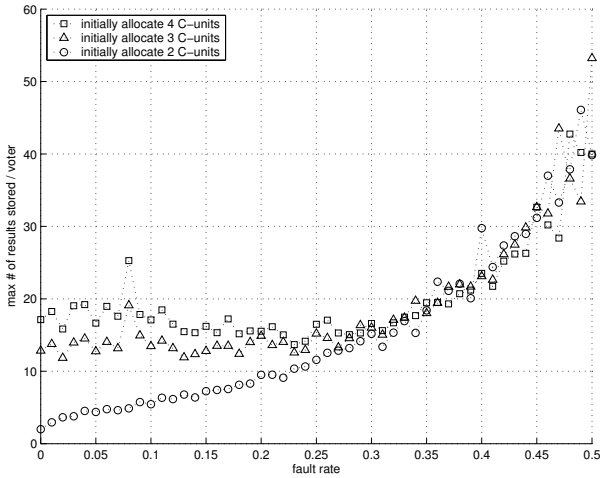

Figure 12: Peak Voter HW comparison for various initial NMR #

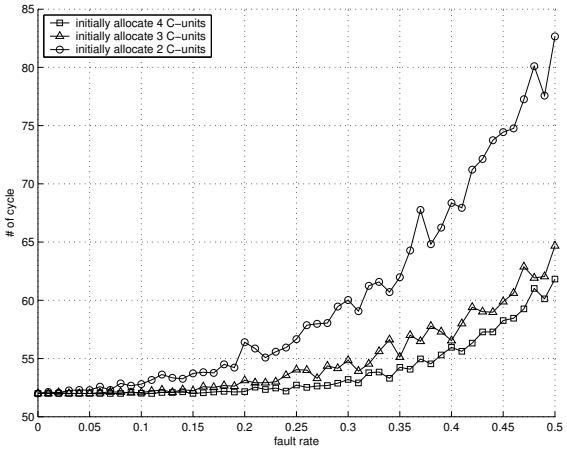

Figure 13: Latency comparison for various initial NMR #

terms of hardware. The circled line in the figures represents the configuration of initially allocating 2 C-units, the line marked with triangles represents the 3 initial C-units case while the line marked with squares shows the case of 4 initial C-units. The fault rates examined in this set of experiments range from 0 to 0.5.

It can be seen from figure 13 that, when initially allocated more C-units, the total latency for the instructions is significantly improved in the high fault rate range of 0.3 to 0.5.

Figures 9 and 10 show the hardware resource comparison for the different configurations in terms of average and maximum number of C-units utilized per cycle. Figures 11 and 12 show the comparison of average and maximum storage requirement for a voter in the different configurations. These data consistently indicate that in terms of hardware consumption, different numbers of initially allocated C-units show insignificant differences when the fault rate is high (0.3-0.5). However, when the fault rate is low (0 - 0.3), the hardware resource consumption is strongly related to the number of initially allocated C-units, where a smaller setting always consumes less amount of hardware.

The comparison of the various configurations of the proposed scheme shows that, even within the proposed strategy, multiple tradeoff points exist. Essentially, when the fault rate is low, the configuration of minimum initial C-unit allocation provides a highly attenuated loss of latency, while exhibiting a significantly reduced amount of hardware consumption. When the fault rate is high, allocating more C-units in the initial cycle helps reduce the overall latency by and large, while consuming almost the same hardware resources as the minimum initial C-unit allocation configuration. These experimental results thus show how the initial C-unit allocation number in the proposed scheme provides different optimal points under various fault rate ranges.

5. CONCLUSION

In this paper, we have provided a computational model for the emerging nanoelectronic environment, where the main issues include reliable computation and system performance. The main techniques in the proposed scheme include: 1) a fault tolerance computation scheme exploiting hardware and time redundancy to guarantee the correctness of each instruction, 2) the idea of computation through multiple speculative branches to improve system performance in data dependent instructions, and 3) the algorithm of dynamically allocating computation units to avoid exponential growth in hardware consumption.

The proposed processor architecture exploits the abundant hardware resources provided by the nanoelectronic technology and makes tradeoffs between hardware and computation performance. Fault tolerance in pipelined instruction execution can be guaranteed and system performance is boosted as well. Through the simulation of different parameters for the proposed scheme, several tradeoff points are identified, providing an insight in selecting the proper parameter for a certain fault rate range to achieve the best hardware/latency tradeoff. The overall experimental results confirm the effectiveness of the proposed scheme from both a performance and a hardware overhead perspective, thus evincing that the proposed approach provides a strong solution to the vital challenges in architecture level fault tolerant computation in nanoelectronic processors.

6. REFERENCES

[1] P. Mazumder, S. Kulkarni, M. Bhattacharya, J. P. Sun and G. I. Haddad, "Digital Circuit Applications of Resonant Tunneling Devices", *Proceedings of the IEEE*, vol. 86, n. 4, pp. 664–686, April 1998.

[2] C. S. Lent, P. D. Tougaw, W. Porod and G. H. Bernstein, "Quantum Cellular Automata", *Nanotechnology*, vol. 4, pp. 49–57, 1993.

[3] Y. G. Krieger, "Molecular Electronics: Current State and Future Trends", *J. Structural Chem*, vol. 34, pp. 896–904, 1993.

[4] ITRS, *International Technology Roadmap for Semiconductors Emerging Research Devices*, 2004.

[5] European Commission, *Technology Roadmap for Nanoelectronics*, 2001.

[6] M. S. Montemerlo, J. C. Love, G. J. Opitech, D. G. Gordon and J. C. Ellenbogen, *Technologies and Designs for Electronic Nanocomputers*, MITRE, July 1996.

[7] P. Beckett and A. Jennings, "Towards Nanocomputer Architecture", in *Asia-Pacific Computer System Architecture Conference*, pp. 141–150, 2002.

[8] J. R. Heath, P. J. Kuekes, G. S. Snider and S. Williams, "A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology", *Science*, vol. 280, pp. 1716–1721, June 1998.

[9] S. C. Goldstein and M. Budiu, "NanoFabrics: Spatial Computing Using Molecular Electronics", in *ISCA*, pp. 178–191, 2001.

[10] S. C. Goldstein, M. Budiu, M. Mishra and G. Venkataramani, "Reconfigurable Computing and Electronic Nanotechnology", in *ASAP*, pp. 132–143, 2003.

[11] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components", in C. Shannon and J. McCarthy, editors, *Automata Studies*, Princeton University Press, 1956.

[12] J. Han and P. Jonker, "A System Architecture Solution for Unreliable Nanoelectronic Devices", *IEEE Transactions on Nanotechnology*, vol. 1, n. 4, pp. 201–208, December 2002.

[13] K. Nikolic, A. Sadek and M. Forshaw, "Architectures for Reliable Computing with Unreliable Nanodevices", in *IEEE-NANO*, pp. 254–259, 2001.

[14] M. A. Gomaa, C. Scarbrough, T. N. Vijaykumar and I. Pomeranz, "Transient-Fault Recovery for Chip Multiprocessors", *IEEE Micro*, vol. 23, n. 6, pp. 76–83, November/December 2003.

[15] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design", in *ACM/IEEE Annual Symposium on Microarchitecture*, pp. 196–207, 1999.

[16] D. K. Pradhan and N. H. Vaidya, "Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture", *IEEE Transactions on Computers*, vol. 43, pp. 1163–1174, October 1994.

[17] B. Izadi and F. Ozguner, "Enhanced Cluster k-Ary n-Cube, A Fault-Tolerant Multiprocessor", *IEEE Transactions on Computers*, vol. 52, n. 11, pp. 1443–1453, November 2003.

[18] W. Rao, A. Orailoglu and R. Karri, "Fault Tolerant Nanoelectronic Processor Architectures", in *ASPDAC*, pp. 311–316, 2005.