Fault Tolerant Nanoelectronic Processor Architectures *

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

In this paper we propose a fault-tolerant processor architecture and an associated fault-tolerant computation model capable of fault tolerance in the nanoelectronic environment that is characterized by high and time varying fault rates. The proposed fault tolerant processor architecture not only guarantees the correctness of computation but also is flexible in that it dynamically trades-off computation resources and performance. The core of the architecture is a decentralized instruction control unit called the voter that achieves both fault tolerance and the maximum parallel execution of instructions by exploiting the abundant computational resources provided by nanotechnologies. Although the result of each instruction needs to be confirmed by executing it on multiple computation units, multiple unconfirmed instructions can proceed as speculative branches. The voter implements a hardware-frugal computation unit allocation algorithm to organize the redundant computations and to dynamically control the growth of speculative branches.

1. INTRODUCTION

New technologies based on nano-scale physical characteristics such as Resonant Tunneling Diodes [1], Quantum-dot Cellular Arrays [2] and molecular electronics [3] have been researched and are being proposed as candidates for next generation device technologies. Due to their nano-scale physical characteristics, these emerging device technologies are anticipated to have drastically improved operating speeds, ultra low power consumption and device densities reaching 10^{12} device / cm^2 [4]. However, system performance is influenced by other factors such as instruction level parallelism, data dependency between instructions, the contention for the computational units and so on.

Physical limitations at the nano-scale result in highly unreliable nano devices thereby making fault tolerance an important design objective. The fault rates in these emerging nanotechnologies are projected to be in the order of $10^{-3} - 10^{-1}$. Furthermore, the faulty behavior is time varying and hard to model [4, 5, 6]. Such fault behavior can be compared with the static fault rates of $10^{-9} - 10^{-7}$ in CMOS technology.

Related research in architectural-level fault tolerance schemes for processors based on CMOS technology include [7, 8, 9, 10]. However, these schemes deal with a low and relatively fixed fault rate and are not applicable to the nanoelectronic environment. Related research in fault tolerance in the nanoelectronic environment includes the Teramac reconfigurable system [11] and the embryonics system [12]. These approaches reconfigure the redundant hardware to bypass the permanent faulty units. The N Module Redundancy and NAND multiplexing [13, 14] use majority vote among replicated hardware to implement fault tolerance at the logic gate level. These simple hardware redundancy based schemes at the logic and lower levels require an immense number of redundant computation units to tolerate the large failure rates in these emerging nanotechnologies [15].

A more powerful fault tolerance scheme supported by a computation mechanism in the processor architecture is therefore required for the nanoelectronic environment. The emerging nanotechnologies can support a large number of high-level computation units, which can be exploited to improve system performance while providing a powerful fault tolerance scheme that can dynamically trade-off time redundancy and hardware redundancy to efficiently guarantee the correctness of computations.

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 the 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 investigate the characteristics of nanoelectronic technologies and develop a fault tolerant processor architecture for high performance fault tolerant computation. We propose a *voter/C-unit* hierarchical structure, which not only supports decentralized control on the dispatch, execution and confirmation of instructions to eliminate contention during instruction processing, but also exploits the abundant computational resources to maximize the number of instructions that can be executed in parallel. Correctness of every instruction is confirmed by multiple execution instances. We develop an algorithm for redundant instruction execution that dynamically manages the hardware and time redundancy assigned to each instruction. The algorithm guarantees correctness of instruction execution, allows multiple speculative execution branches to proceed and dynamically manages the hardware consumption. Overall, the proposed strategy trades off hardware and time redundancy dynamically to improve both fault tolerance and system performance.

We first motivate the proposed scheme in section 2, then provide the algorithm together with an example in section 3. In section 4, a probabilistic analysis is provided to illustrate the effect of the proposed scheme. A possible architectural framework for the proposed scheme is provided in section 5 while section 6 concludes the paper.

2. MOTIVATION

We motivate the proposed approach in this section by investigating a sequence of conflicts among hardware resources, system performance and fault tolerance in a nanoelectronic environment where fault rates are high and time varying.

Consider a straightforward N modular hardware redundancy (NMR) fault tolerance strategy. N copies of an instruction are executed by N distinct units in parallel and the result is confirmed by a majority vote. At first glance, this is an appealing strategy since the 10^{12} device/cm^2 densities can support quite a large number of computation units. A careful analysis reveals however that this approach is practical only

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

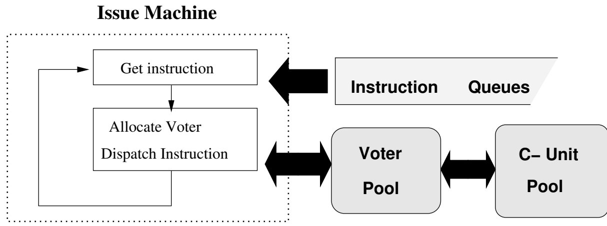

Figure 1: Instruction issue process with voter/C-unit structure

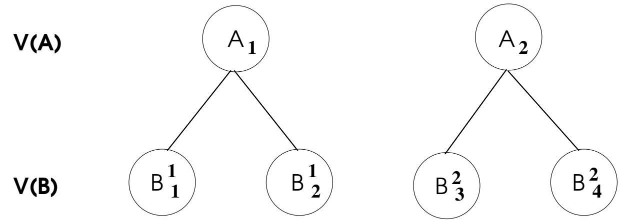

Figure 2: An example of computation forest

if the fault rates are low or steady enough to yield confirmed results. However, since in the emerging nanotechnologies the fault rates are in contrast quite high and time varying, the inflexibility of the NMR fault tolerance strategy limits its applicability severely.

Consider a straightforward time redundancy fault tolerance strategy. Multiple clock cycles may elapse before the result of an instruction can be confirmed. This severely compromises system performance since subsequent instructions contend for the instruction control unit and need to wait. The centralized instruction control unit in traditional processor architectures thus becomes the bottleneck for system performance. In fact, in the nanoelectronic environment, the abundant hardware resources can be exploited to implement a decentralized control mechanism for instructions using a large number of decentralized control units. Therefore, instructions can be issued without stalls.

To deal with time varying fault rates, time redundancy can be utilized in a manner complementary to hardware redundancy. When the results generated by N computation units cannot confirm a result, the initial hardware redundancy approach fails. A control unit can then dynamically collect additional units to repeat the instruction in the next cycle. This process can be repeated until the result of the instruction is confirmed. With this additional time redundancy, system reliability can be guaranteed even when the fault rate is time varying. The flexibility inherent in this hybrid technique necessitates design of a control unit that dynamically selects between hardware and time redundancy.

Data dependencies among instructions also introduce delays. Time redundancy can further deteriorate these delays by prolonging confirmation of instructions. 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 depends on the result of this instruction is then delayed, resulting in a domino effect on subsequent dependent instructions.

To solve this problem, it can be observed that a dependent instruction need not wait for the confirmation of its predecessor results; additional units can be used to speculatively execute an instruction. A dependent instruction can use unconfirmed results in a speculative manner. Multiple speculative branches may be formed for a dependent instruction. As results are confirmed, the correct branches of the dependent instruction are retained and the remaining branches are pruned. While speculation can speed up instruction execution especially in the presence of data dependencies, one has to carefully manage the amount of speculation. Speculative branches can grow exponentially and quickly exhaust even the abundant hardware available in a nanoelectronic environment. We will present an allocation algorithm that carefully manages speculative instruction execution by allocating hardware frugally.

3. FAULT TOLERANT COMPUTATION IN NANOELECTRONIC PROCESSORS

In a nutshell, the key features of the proposed high-performance fault-tolerant nanoelectronic processor architecture are: (i) a large number of decentralized instruction control units (voters), (ii) a large number of complex computation units (C-units), (iii) support for speculative instruction execution of dependent instructions, and (iv) a hardware-frugal dynamic C-unit allocation algorithm.

3.1 Voter/C-unit structure

In the nanoelectronic environment where device density is high, a nanoelectronic processor can have a large number of computation units available. Therefore, parallel execution of instructions constitutes a viable technique for improving system performance. However, the parallel execution of instructions demands not only multiple execution units, but also decentralized control units. In the proposed architecture for nanoelectronic processors, a pool of decentralized control units called *voters* that support the parallel execution of instructions are presumed to exist.

When an instruction is issued, a voter is allocated to it. The voter manages the execution of the instruction and is responsible for returning a confirmed result. Meanwhile, the issuing machine continues to process the next instruction in the instruction queue without delay.

Computation units (*C-units*) are dynamically allocated and released by a voter. To confirm the result of an instruction allocated to it, the voter carries out the same instruction on multiple C-units and compares their results. Figure 1 shows a high-level view of the instruction issue process and the interaction between the voters and the C-units.

The voter accomplishes fault tolerance computation by combining hardware and time redundancy. Initially, the voter allocates two C-units for an instruction using hardware redundancy. If the two results agree, then the instruction is deemed confirmed. Otherwise, the voter incrementally applies time redundancy by allocating one C-unit at a time until two of the results agree and the instruction can be confirmed. Triple modular redundancy (TMR) is not necessary for this initial hardware redundancy, since a follow-on time redundancy can be invoked if an instruction cannot be confirmed in the first cycle.

3.2 Computing with speculative branches

In order to improve system performance, in the proposed fault tolerant nanoelectronic processor architecture, dependent instructions speculatively use the result of an as yet unconfirmed instruction. Dependent instructions do not stall to await confirmation of their operands. As comparison information is obtained by the voter, it is used to either confirm or prune the speculative branches of dependent instructions.

Consider an instruction B that is data dependent on instruction A as shown in Figure 2. The voter $V(A)$ that manages the execution and confirmation of A allocates two C-units, denoted as A_1 and A_2, to execute the two instances of A. When instruction B is issued, the results of A_1 and A_2 have not been compared; therefore A is not yet confirmed. Hence in this example voter $V(B)$ allocates four C-units to speculatively compute B. The speculative branches B_1^1 and B_2^1 take the result from A_1 while the speculative branches B_3^2 and B_4^2 take the result from A_2.

For representational convenience, we use superscripts of a C-unit to indicate the parent C-unit from which an unconfirmed result is taken; we use subscripts of a C-unit for its own index. The C-units of dependent instructions form a *computation forest*, where a child node always takes the result from its parent node. The information of speculative branches is maintained and managed by a voter. It is used to confirm or prune the speculative branch when confirmation arrives from parent level instructions.

3.3 Dynamic hardware allocation algorithm

In the example of figure 2, the voter of B allocates two C-units for each speculative instruction. This strategy assumes that the unconfirmed results from the parent level will be distinct. Based on this pessimistic estimation of the fault rate, a voter needs to allocate twice as many C-units as are allocated at the parent level of the computation forest. Therefore, hardware resources are allocated at an exponential rate when a sequence of dependent instructions is issued. This might prematurely exhaust available C-units before confirming instructions and pruning erroneous speculative branches, existing abundant hardware resources in the nanoelectronic processor architecture notwithstanding.

The assumption that every speculative branch generates a distinct result is apparently an overestimation of the fault rate. Based on a more realistic estimation of the fault rate, we propose a flexible hardware allocation strategy that always assumes the speculative branches to produce the same result, if they use the same inputs. For instance, in the example of figure 2, the proposed allocation strategy assumes that $A_1 = A_2$ and thus will allocate only two C-units to B, each taking the result from one C-unit used by A.

Basically, the proposed strategy always allocates the least number of hardware resources for the speculative instructions as long as there exist possibilities to confirm the result. Only when concrete information shows that it is no longer possible to confirm an instruction, additional hardware is allocated. The following sections describe the strategy based on the concept of a *confirmation group*.

3.3.1 Confirmation group

A confirmation group is defined as a set of C-units allocated to an instruction that might generate an identical result. In the example of figure 2, $\{A_1, A_2\}$ is a confirmation group before the comparison of their results is available. Similarly, $\{B_1^1, B_2^1, B_3^2, B_4^2\}$ is the confirmation group for B. If two C-units disagree, then at most one of them is correct; hence the concept of confirmation group essentially represents mutually exclusive speculative branch groups.

When the C-units in a confirmation group yield different results, the voter splits the confirmation group. For example, if $A_1 \neq A_2$, then the voter $V(A)$ splits the confirmation group $\{A_1, A_2\}$ into two confirmation groups $\{A_1\}$ and $\{A_2\}$. In turn, the confirmation group of instruction B is split by $V(B)$ into $\{B_1^1, B_2^1\}$ and $\{B_3^2, B_4^2\}$. Since instruction A cannot be confirmed, voter $V(A)$ needs to allocate a new C-unit A_3 to execute A. This results in confirmation groups $\{A_1, A_3\}$ and $\{A_2, A_3\}$, because it is still possible for the members of each confirmation group to generate identical results.

Based on the confirmation group concept, an instruction can be in one of four states: not confirmable, confirmable, locally confirmed and confirmed.

- When none of the confirmation groups of an instruction contains two or more C-units, the instruction is *not confirmable*. One cannot find two C-units whose results match to confirm this instruction since C-units in different confirmation groups do not agree. If an instruction is *not confirmable*, more C-units need to be allocated by the voter. In the example of figure 2 if $A_1 \neq A_2$, then $\{A_1\}$, $\{A_2\}$ become the two confirmation groups, each containing a single C-unit. Instruction A is *not confirmable* and the voter needs to allocate a new C-unit, A_3.
- When there is at least one confirmation group containing more than one C-unit, the instruction is *confirmable*. Under this situation, if the two C-units in the same confirmation group generate identical results and this speculation branch is confirmed to be correct, the instruction can be confirmed.
- If two C-units in a confirmation group yield the same result, the corresponding instruction can be *locally confirmed*. Furthermore, if the parent instruction confirms the branch of speculative instructions to be the correct branch, then the instruction becomes *confirmed*. If $B_1^1 = B_2^1$, then instruction B is *locally confirmed*. When $A_1 = A_3$, A_1 is *confirmed* and in turn the status of instruction B changes from *locally confirmed* to *confirmed* with the identical result of B_1^1 and B_2^1.

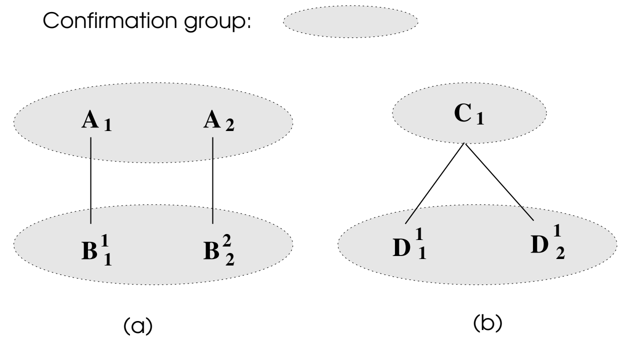

Figure 3: C-unit initial allocation cases

When a voter confirms an instruction, it passes the information to voters assigned to its dependent instructions so that correspondent speculative branches can be pruned.

3.3.2 Hardware-frugal C-unit allocation algorithm

Since an instruction can be confirmed when the results of two C-units agree, the allocation strategy always limits the number of elements in a confirmation group to be at most two. If an instruction does not depend on the results of any other instructions, the C-unit allocation strategy is straightforward:

1. Initially allocate two C-units as a confirmation group.
2. If the C-units in a confirmation group produce distinct results, repeat the following steps until the instruction is confirmed:
   (a) Split the confirmation group.
   (b) Allocate a new C-unit and update the confirmation groups.
3. Generate the confirmed result and inform voters of successor dependent instructions

If an instruction depends on the result of another instruction, the associated voter needs to allocate C-units according to the confirmation groups of the instruction that it depends on. If instruction B depends on the result of A, the confirmation groups of A spawn mutually exclusive speculative branches. The voter $V(B)$ should allocate C-units frugally so as to increase the likelihood of confirming B. When instruction B is issued, $V(B)$ performs an initial allocation of C-units for B as follows:

- For each confirmation group in A:
  - If the confirmation group contains two elements, then allocate one C-unit taking the result from each element, as is shown in figure 3 (a).
  - If the confirmation group contains only one element, then allocate two C-units, both taking the result from the same element. This is shown as $\{D_1^1, D_2^1\}$ in figure 3 (b).

When information is passed from $V(A)$ or the status of B changes, the algorithm performs adjustments for allocations as follows:

- If A is confirmed, keep the C-units of B in the correct speculative branch and prune the incorrect branches.
  - If the confirmation group of the correct branch in B is already locally confirmed, then confirm instruction B.

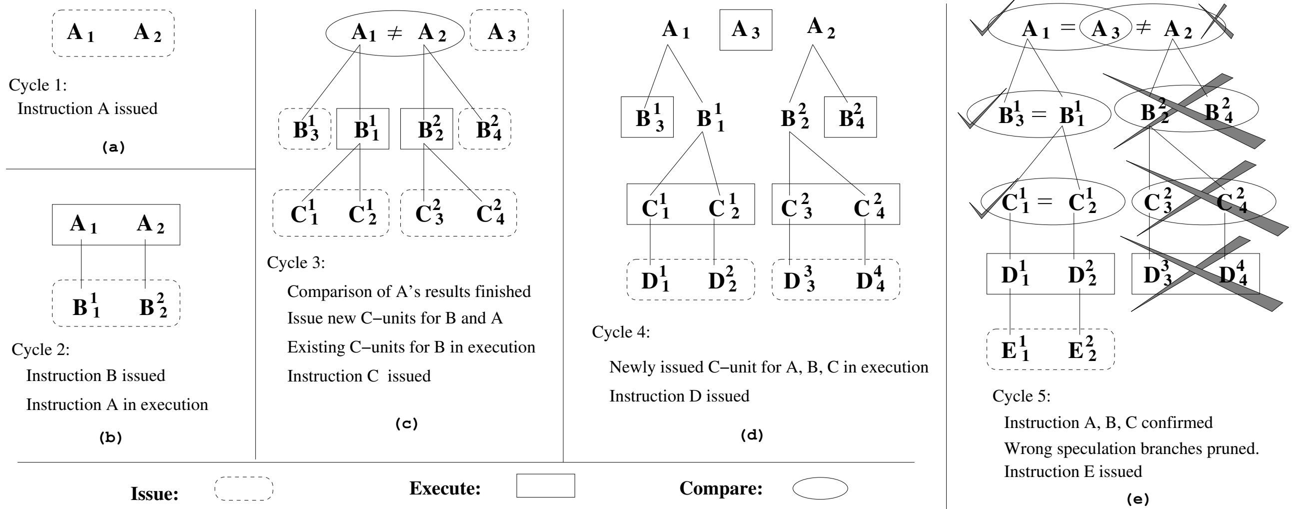

Figure 4: An example of five cycles for a sequence of instructions using the proposed allocation algorithm

- If a confirmation group in A is split, then split their successor instruction B's confirmation group accordingly.
- If a new confirmation group is formed in A, while no C-units in B are to receive its results, then allocate two C-units in B to receive the results of the newly formed speculative branch in A.
- If instruction B becomes *not confirmable*, i.e., all the confirmation groups contain only one element, then allocate new C-units so that every confirmation group contains two elements.

According to the algorithm, every instruction is initially allocated two C-units in each confirmation group; thus when no fault occurs, confirmation can be attained with minimum hardware resources in the shortest time. It is to be noted that two C-units are allocated per *confirmation group* and not per speculative branch. This approach avoids the doubling in the number of C-unit allocations for every dependent instruction. The adjustment allocation algorithm constrains the growth of the speculative branches by allocating new C-units only when the instruction is not confirmable.

As a result, when the fault rate is low, the algorithm uses a minimum number of C-units and can achieve high system performance. When the fault rate is high, the C-units are allocated on an as necessary basis and time redundancy is exploited to guarantee the confirmation of instructions.

3.4 An example

We show an example of dynamic allocation of C-units in figure 4, where a sequence of instructions, A, B, C, D, E, are executed and confirmed, each depending on the result of the previous one. A C-unit of any instruction can be in one of the three main states: issue, execute and compare. At the issue state, the C-unit is allocated by the voter and assigned operands for the instruction. At the execution cycle, a C-unit carries out the actual computation. At the confirm state, the result of a C-unit is returned to its voter and the voter makes further decisions according to the comparison of results.

Figure 4(a) shows the first cycle, during which instruction A is being issued; the voter $V(A)$ initially allocates two C-units, A_1 and A_2, for its execution in this cycle. In the second cycle shown in the (b) part, A_1 and A_2 are in the execution cycle, while at the same time instruction B is issued. B depends on A's result. At this stage there is only one confirmation group in A, which contains the two C-units A_1 and A_2. According to the algorithm, $V(B)$ allocates two C-units B_1^1 and B_2^2, taking the results from A_1 and A_2 accordingly.

In the third cycle, which is shown in figure 4(c), the comparison result of A_1 and A_2 is available and found to be in disagreement. The one confirmation group of A is thus split into two, each containing one element. The confirmation group in B is also split due to the different results of A_1 and A_2.

Since A becomes unconfirmable, $V(A)$ allocates a new C-unit A_3 and issues it for instruction A. $V(B)$ also allocates two new C-units and issues them for instruction B so that each confirmation group can have two elements. The two confirmation groups in B are now $\{B_3^1, B_1^1\}$ and $\{B_2^2, B_4^2\}$.

Instruction C is issued in this cycle. For each confirmation group of B, two C-units are allocated. However, C_1^1 and C_2^1 are set to take the result from B_1^1, while C_3^2 and C_4^2 are set to take the result from B_2^2. This is because B_1^1 and B_2^2 are in the execution stage so their results will be available in the next cycle. On the other hand, B_3^1 and B_4^2 are still in the issue stage, so no C-units are allocated to await their result at this cycle.

The (d) part of figure 4 shows the fourth cycle of the example. A_3, B_3^1 and B_4^2 are now in the execution stage while the results of B_1^1 and B_2^2 are available and have been passed to $\{C_1^1, C_2^1\}, \{C_3^2, C_4^2\}$ accordingly. No comparison is made between B_1^1 and B_2^2 because they belong to distinct confirmation groups. All the four C-units of instruction C are in the execution stage. Instruction D is issued with four C-units allocated into two confirmation groups.

The (e) part in figure 4 shows the propagation of confirmation as well as the branch pruning process in the fifth cycle. In this cycle, instruction A is confirmed with $A_1 = A_3$. Therefore, A_2 is identified as an incorrect speculative branch and gets pruned. Also in this cycle, instructions B and C are locally confirmed with $B_3^1 = B_1^1$ and $C_1^1 = C_2^1$. Since the speculative branch of A_1 is confirmed to be correct, instruction B is confirmed, which further confirms instruction C.

The example shows that the proposed hardware frugal allocation algorithm can be used to achieve a fault tolerance scheme with high flexibility and performance. In this example, although A_2 is faulty and the confirmation of A is delayed for two cycles, with the proper control of speculative branches, the delay is not propagated to the subsequent instructions and the whole sequence of instructions can be confirmed without delay.

4. PROBABILISTIC ANALYSIS

By providing a mechanism to arrange the dependent instructions to execute as speculative branches instead of waiting for the confirmation, the proposed scheme improves performance to a large extent. When the fault rate is low, a computation forest with a depth of d takes $(d + 1)$ cycles to finish with the proposed scheme. However, for the

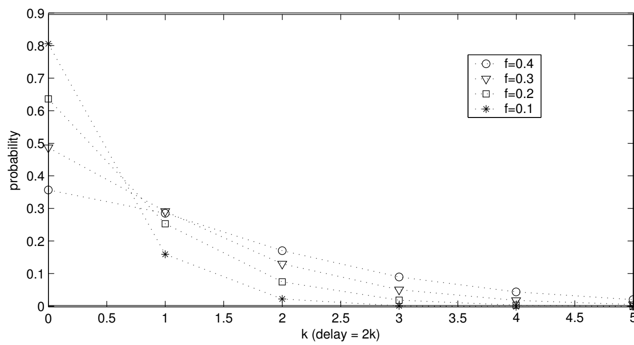

Figure 5: The probabilistic delay distribution of an instruction in the root level of a computation forest

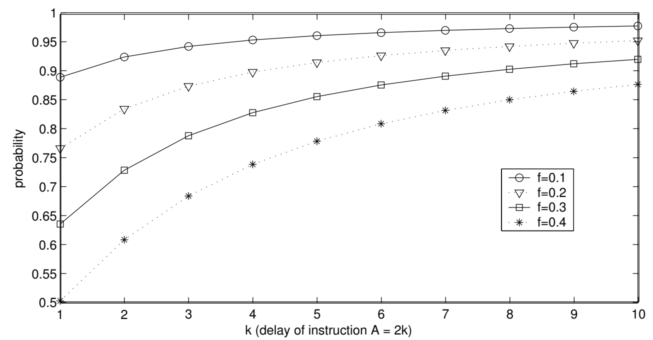

Figure 6: The probability of a dependent instruction being able to finish one cycle ahead

simple scheme where every instruction waits for the confirmation of the instruction it depends on, at least $2d$ cycles are required.

Figure 5 shows a delay probability for one single instruction A with no dependency on other instructions under various fault rates. The k axis denotes the number of delayed cycles needed to confirm A, where the delay is always $2k$. The Y axis denotes the probability. Assuming a fault rate of f, the proposed scheme allocates two C-units, say A_1 and A_2, initially to compute A. The probability of the instruction being able to finish without delay is essentially the probability of $A_1 = A_2$; thus, it is $(1-f)^2$. When $A_1 \neq A_2$, another C-unit, A_3, is allocated. If A_3 equals either A_1 or A_2, then the instruction can be confirmed with a delay of two cycles, one for calculating A_3 and one for comparing A_3 with the available results. The associated probability is $2(1-f)^2 f$. Similarly, when A_4 allocated and equals one of (A_1, A_2, A_3), then the instruction can be confirmed with a delay of four cycles, with the probability $3(1-f)^2 f^2$. Essentially, the instruction is always delayed for an even number of cycles and the probability of the instruction being confirmed with a delay of $2k$ cycles is $(k+1) f^k (1-f)^2$.

The delay for an instruction can actually get amortized when there is a sequence of dependent instructions, i.e., when the depth of the computation forest is more than one. In figure 6 we show the probability of a subsequent dependent instruction B being able to reduce the delay produced by A for one cycle. The k axis denotes the latency to confirm A, which is always $2k$ cycles; the Y axis shows the probability of instruction B being able to be confirmed at the same cycle as A, where B depends on A.

Since B is issued one cycle later than A, its confirmation should be one cycle after A's confirmation as well. Therefore, when the confirmation of A is delayed for $2k$ cycles, B's result is expected to be confirmed with at least a delay of $2k$. However, with the speculative branches computing ahead of confirmation, in most cases, B can be confirmed at the same cycle as A, i.e., B's confirmation is delayed for only $(2k-1)$ cycles. Therefore, it is possible for B to reduce the delay caused by A for one cycle.

This is shown in the example of the previous section, where not only B, but also the subsequent instruction C is confirmed at the moment A is confirmed. Therefore, the delay of a sequence of dependent instructions can be significantly shortened.

To calculate the probability of B being confirmed at the same cycle as A in the case of A having a delay of $2k$, we start with an example of $k = 2$. When no agreeing pair of results can be attained within (A_1, A_2, A_3) but A_4 equals one of them, A is delayed for 4 cycles. This is because the execution and comparison of A_3 takes two cycles, and the same happens with A_4; thus $k = 2$ and the delay is $2k = 4$.

According to the proposed algorithm, $\{B_1^1, B_3^1\}$ take the result from A_1, while $\{B_2^2, B_4^2\}$ take the result from A_2. The confirmations of $\{B_1^1, B_3^1\}$ and $\{B_2^2, B_4^2\}$ are performed in the same cycle as comparing A_3 with A_1 and A_2. Since A_3 does not agree with A_1 or A_2, a new confirmation group in A is formed as $\{A_3\}$. According to the algorithm, at the same time A_4 is issued, $\{B_5^3, B_6^3\}$ are issued taking A_3's result. The comparison of B_5^3 and B_6^3 is performed at the same cycle as comparing A_4 to others. Therefore, if the correct result is A_3, i.e., $A_3 = A_4$, then the branch $\{B_5^3, B_6^3\}$ is valid. In this situation, if $B_5^3 = B_6^3$, then B can be immediately confirmed at the same cycle as A. On the other hand, if $B_5^3 \neq B_6^3$, i.e., B introduces a delay by itself, then B will not be able to finish at the same cycle as A.

If the correct result of A is A_1 (or A_2), then even if B produces two cycles of delay, it is still able to be confirmed at the same cycle as A. This is because in this situation, the local confirmation of the correct branch $\{B_1^1, B_3^1\}$ is 2 cycles ahead of the confirmation of A; thus the two-cycle slot can be used as a cushion to absorb a two-cycle local delay of B without infecting the final confirmation.

In summary, the probability of B being able to reduce the delay of A by one cycle is the sum of the following probabilities:

- $A_4 = A_1$ or A_2, B has 0 or 2 local delay;
- $A_4 = A_3$, B has 0 local latency;

In the general case where A has a delay of $2k$, the final confirmation achieved through the comparison of A_{k+2}, the probability of B being able to reduce the delay of A by one cycle, is the sum of the following probabilities:

- $A_{k+2} = A_1$ or A_2, B has 0 or up to $2k-2$ local delay;
- $A_{k+2} = A_3$, B has 0 or up to $2k-4$ local delay;
- $A_{k+2} = A_4$, B has 0 or up to $2k-6$ local delay;

  ...
- $A_{k+2} = A_k$, B has 0 or up to 2 local delay;
- $A_{k+2} = A_{k+1}$, B has 0 local delay;

From the results shown in figure 6, we can observe that even under a dramatically high fault rate ($f = 0.4$), the probability that a subsequent dependent instruction can reduce the delay of its parent instruction by one cycle is quite high (0.5). The probability increases when the number of delays in instruction A is larger.

If the confirmation of A delays for multiple cycles and there are a sequence of dependent instructions such as B, C, D..., the above analysis shows that with very large probability B can reduce the delay by one cycle. Furthermore, C can provide a two-cycle reduction for the delay, while D can provide a three-cycle reduction. Therefore, even if an instruction can have a long delayed confirmation process, possibly as a result of the clustered fault behavior in the nanoelectronic environment, the proposed scheme is able to significantly reduce the delay for its subsequent dependent instructions.

5. PROCESSOR ARCHITECTURE

In this section we illustrate a possible implementation of the nanoelectronic processor architecture for the proposed fault tolerance scheme. Typically, a post-fabricated nanoelectronic system is expected

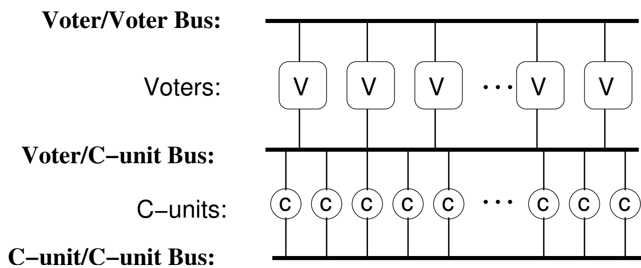

Figure 7: The three-bus architecture

to have a grid structure with high regularity and abundance of reconfigurable computation units, each embedded with configurable computation capability and some memory units [4, 16, 17]. In this section, we assume each C-unit as well as voter can be constructed by configuring a computation grid cell in the nanoelectronic system after fabrication.

5.1 Voter functionality

When an instruction is issued, it is assigned a specific voter, which takes over the responsibility of the correct execution of the instruction. In the issue cycle, the voter allocates C-units according to the proposed algorithm and then passes the parameters of the instruction to the C-units for execution. If the instruction depends on the result of another one, the voter also registers a link on each C-unit, indicating from which parent C-unit the data should be expected. Therefore, the passing of results in the computation forest between dependent instructions can be effected directly from C-unit to C-unit with no voter involvement.

The voter retains the information of all the C-units allocated for the instruction and all the confirmation groups of the instruction. The voter also retains the results returned from the C-units; thus the C-units can be released once the computation is finished.

After comparing the results from the C-units, a voter makes a decision as to whether to confirm the computation or to keep allocating new C-units. A voter communicates with the voter of the upper level instruction, so as to dynamically manage the C-units and perform updates on confirmation groups. A voter also transfers the confirmation information to lower level voters in the computation forest, thus pruning the incorrect speculative branches.

5.2 Bus structure

Similar to the bus structure in Tomasulo algorithm [18], the proposed scheme can utilize buses to listen to and pass messages efficiently. In order to eliminate bus contention so as to improve performance, three types of buses can be used according to the types of messages passing in the system:

- **voter/voter:** for messages of confirmation group split and the pruning of speculative branches.
- **voter/C-unit:** for the C-unit allocation/release message and the reporting of results from a C-unit to its voter.
- **C-unit/C-unit:** passing results between dependent instructions

In figure 7, the structure of the three types of buses is shown. The Voter/Voter bus connects all the voters, the Voter/C-unit bus connects voters to the C-units while the C-unit/C-unit bus connects all the C-units. Through the utilization of dedicated buses for different types of messages, the contention on a common bus is alleviated.

6. CONCLUSION

In this paper, we provide 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 a decentralized control implemented by the voter/C-unit hierarchical structure, the idea of computation through multiple speculative branches and the algorithm of dynamically allocating computation units. The proposed processor architecture exploits the abundant hardware resources provided by the nanoelectronic technology and makes trade-offs between reliability and computation performance. Therefore, fault tolerance in instruction execution can be guaranteed and system performance is boosted as well. The proposed scheme thus provides a possible solution to the important problem of fault tolerant computation in the nanoelectronic architecture.

7. 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] European Commission, *Technology Roadmap for Nanoelectronics*, 2001.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] D. Mange, M. Sipper, A. Stauffer and G. Tempesti, “Toward Robust Integrated Circuits: The Embryonics Approach”, *Proceedings of the IEEE*, vol. 88, n. 4, pp. 516–541, April 2000.

[13] 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.

[14] 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.

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

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

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

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, Third Edition, 2002.