Note: The final version may be slightly different from this copy.

Towards Achieving Reliable and High-Performance Nanocomputing via Dynamic Redundancy Allocation

Shuo Wang, Lei Wang, and Faquir Jain
Department of Electrical and Computer Engineering
University of Connecticut
371 Fairfield Way, U-2157
Storrs, CT 06269-2157, U.S.A.
Email: {shuo.wang, leiwang, fcj}@engr.uconn.edu
Phone: 860-486-3066, Fax: 860-486-2447

Nanoelectronic devices are considered to be the computational fabrics for the emerging nanocomputing systems due to their ultra-high speed and integration density. However, the imperfect bottom-up self-assembly fabrication leads to excessive defects that have become a barrier for achieving reliable computing. In addition, transient errors continue to be a problem. The massive parallelism rendered by nanoscale integration opens up new opportunities but also poses challenges on how to manage such massive resources for reliable and high-performance computing. In this paper, we propose a nanoarchitecture solution to address these emerging challenges. By using dynamic redundancy allocation, the massive parallelism is exploited to jointly achieve fault (defect/error) tolerance and high performance. Simulation results demonstrate the effectiveness of the proposed technique under a range of fault rates and operating conditions.

Categories and Subject Descriptors: B.8.0 [**PERFORMANCE AND RELIABILITY**]: General

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Nanoscale architecture, performance, hardware reliability, redundant design, redundancy allocation

1. INTRODUCTION

Since the invention of CMOS-based integrated circuits (IC), computer system design has reaped a dramatic improvement in computational performance. The key enabling technologies are a combination of advances in semiconductor process and design methodology, and innovations in computer architecture. As con-

Some preliminary results of this work were reported in *International Symposium on Nanoscale Architectures (NANOARCH)*, 2007 [Wang, S. et al. 2007].
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM xxxx-xxxx/20YY/xxxx-0001 $5.00

ventional CMOS technology quickly approaches the end of roadmap, many novel nanoelectronic devices including carbon nanotubes [Martel et al. 2002], silicon nanowires [Huang et al. 2001], quantum-dot cellular automata [Lent et al. 1993; Ma et al. 2008] and resonant tunneling devices [Mazumder et al. 1998] have emerged as the potential computational substrate for nanoscale integration. There are already some work in the literature demonstrating new nanocomputing systems such as programmable logic arrays [Dehon 2005] and application-specific integrated circuits (NASIC) [Wang, T. et al. 2004]. However, the emerging nanoelectronic technology is accompanied by some new challenges that may have a profound impact on architecture-level design and optimization.

It is widely acknowledged that nanoscale integration will no longer enjoy high reliability as the conventional CMOS technology. In comparison with conventional CMOS technology, the defect rates in nanoelectronic devices are projected to be several orders of magnitude higher due to the bottom-up stochastic assembly. In addition, soft (transient) errors continue to be a problem, causing erroneous behaviors that are difficult to model and predict. Thus, building reliable nanocomputing systems from unreliable nanoelectronic devices is becoming a challenging problem that must be addressed at various levels of design hierarchy. At the device/circuit level, defect mapping techniques [Mishra and Goldstein 2003; Tahoori 2005] are proposed to identify defective devices and utilize spare defect-free devices as functional units. However, these per-chip based test-then-reconfigure approaches may become expensive and time-consuming for ultra high-density nanoscale integrated systems. Furthermore, even if defects are detected and mapped out prior to runtime, new defects may still sneak into the system over the life time and transient errors remain unsolved. In another approach, error checking codes (ECC) are widely employed to provide fault tolerance for nanoscale memories and interconnects [Jeffery and Figueiredo 2006; Kuekes et al. 2005], while it might be difficult, if not impossible, for practical ways of implementing ECC in logic operations and instruction processing. At the system level, redundancy based techniques, such as N-modular redundancy and multiplexing logic, were proposed in the pioneering work of John von Neumann [von Neumann 1956] using hardware redundancy to recover from the faults. Recently, NAND multiplexing is extended to a rather low degree of redundancy [Han and Jonker 2002] and new models are presented in [Bhaduri et al. 2007; Roy and Beiu 2005] to study the multiplexing systems. Redundancy based fault tolerance is considered very effective for nanocomputing systems due to the abundant device resources offered by nanoscale integration. Most existing work uses fixed modular redundancy, which lacks adaptivity to different reliability requirements at runtime. An improved strategy [Rao et al. 2005; Rao et al. 2007] was developed to frugally allocate redundancy so as to avoid the resource usage from exponentially increasing.

In nanocomputing systems, the problem of fault tolerance is closely related to high-performance parallel execution. Traditionally, architectural level research for performance improvement has been directed toward exploiting various levels of parallelism. This trend is clearly reflected in simultaneous multithreading (SMT) [Tullsen et al. 1995; Lo et al. 1997] and chip multiprocessors (CMP) [Olukotun et al. 1996; Hammond et al. 1997]. Nanocomputing systems show great

potential to support this trend of parallelism exploitation due to the ultra-high integration density. However, with high defect/error rates, the benefit of parallelism may be limited by the redundant computation necessary for defect/error detection and recovery. Massive parallelism does not necessarily lead to high performance unless the system can effectively recover from unpredictable upsets. Furthermore, the unpredictable upsets also lead to performance unpredictability. This is especially a problem for real-time applications that require stable and predictable execution performance, e.g., multimedia communications that have tight protocol timing specifications. Thus, it is critical to balance the often conflicting requirements on performance and reliability.

The interplay between performance and reliability gives rise to a challenging problem on resource management that is critical to nanocomputing systems. This compels us to explore nanoarchitecture solutions in order to unfold the full potential of nanocomputing paradigm. In this paper, we propose a dynamic redundancy allocation technique that enables reliable and scalable parallelism. The proposed technique manages the parallelism at an optimal level so that both fault tolerance and performance enhancement can be achieved in a coherent manner. Different from the existing redundancy management strategies [Rao et al. 2005; Rao et al. 2007; von Neumann 1956], the proposed technique explicitly considers the availability of the computational resources and the varying requirements on reliability and performance, and therefore is more flexible in redundancy allocation across the instructions at runtime. As the proposed approach can be applied to address both permanent defects and transient errors, we do not differentiate between them and refer the general term *fault* to both.

Some preliminary results were reported in [Wang, S. et al. 2007]. In this paper, we extend our past work by making the following contributions. First, the details of the nanoarchitecture model and localized control mechanism are provided. Second, a systematic performance analysis and design methodology on dynamic redundancy allocation is developed. Third, we conduct a comprehensive study on the application of the proposed technique. Simulation results including the performance predictability demonstrate the advantageous features of dynamic redundancy allocation under a wide range of fault rates.

The rest of the paper is organized as follows. In section 2, we present the nanoarchitecture model and control mechanism as the platform for dynamic redundancy allocation. In section 3, we provide the details of the proposed dynamic redundancy allocation. Simulation results are discussed in section 4 to evaluate the effectiveness of the proposed technique. In section 5, we give the conclusion.

2. NANOARCHITECTURE MODEL

Nanocomputing systems feature highly regular, locally connected, and data parallel architectures that match well to the ultra-high speed and integration density offered by nanoelectronics [Beckett and Jennings 2002; Fountain et al. 1998]. As shown in Fig. 1, the underlying nanoarchitecture may be multiclustered [Franklin and Sohi 1992; Farkas et al. 1997; Goldstein and Budiu 2001], where each cluster consists of multiple functional units that can be tuned specifically for a certain category of applications. The workload is dispatched to the individual clusters using

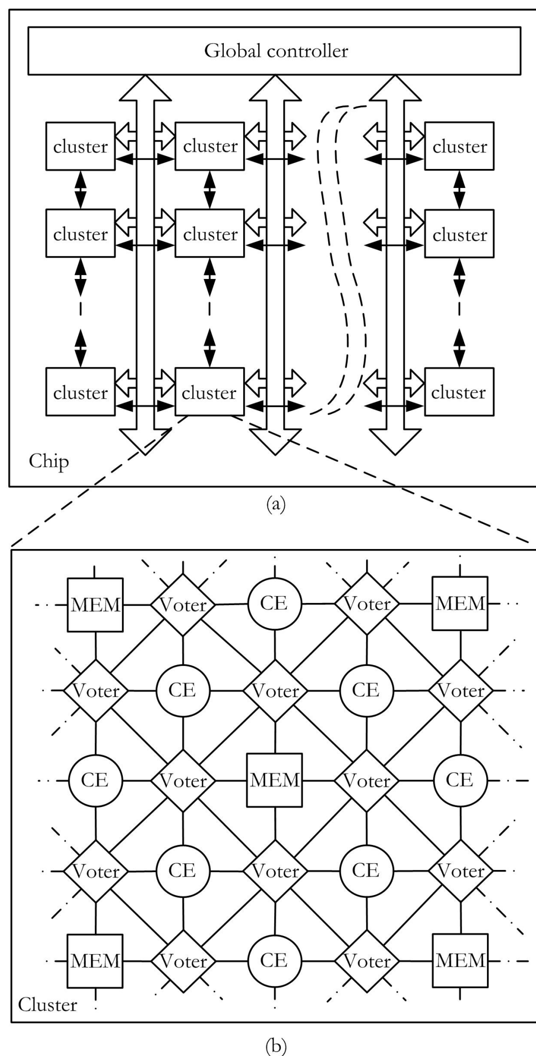

Fig. 1. A conceptual illustration of nanoarchitecture model: (a) top-level view and (b) cluster configuration.

predefined metrics (e.g., slack [Zhu and Fleisch 2000], deadline [He et al. 2004], and mean response time [Tang and Chanson 2001]) to achieve overall performance optimization.

While specific nanocomputing systems may be constructed with various configurations, in this paper we consider a generic nanoarchitecture model where each cluster contains a pool of three basic functional units: computation elements (CE), memory units (MEM), and voters, as shown in Fig. 1(b). The CEs and voters can be implemented by programmable nanoelectronic logic arrays (e.g., nanowire crossbars) according to the defined functions, as shown in [Dehon 2005; Goldstein and Budiu 2001]. Memory units can also be implemented in a similar way [Chen

et al. 2003]. As the computation is inherently fault-prone, it is necessary to execute multiple copies by exploiting the spatial and temporal redundancy in the CEs. The results are then compared by the voters to detect and correct errors based on majority voting. The voters also provide localized control within the cluster by allocating and managing CEs and MEMs according to certain allocation strategies such as the one proposed in section 3. As the control and redundancy allocation are performed distributively, the implementation can be simplified to LUT, as shown in section 3.4. Same as most redundancy-based architectures, the above nanoarchitecture model relies upon the correct voter operation. A complete characterization of reliability issues relevant to voter design needs to be done before the application of the proposed technique. We assume the operation of voters to be correct by using reliability enhancing design techniques. Thus, faults occurred in the voters have a much lower rate than that in the CEs and hence are ignored in the following discussion. In addition, other techniques, e.g., error correcting codes, are effective to deal with faults in interconnect and memory.

To exploit parallel processing, pipelined architecture is implemented in the clusters. For the purpose of demonstration, we consider a prototype pipeline including four stages: *issue*, *execute*, *compare*, and *complete*. Note that we do not lose any generality with this architecture as further split of stages into more complex pipeline is possible depending on the requirement of specific implementations. In the issue stage, voters will be invoked to manage the instruction processing. To facilitate fault tolerance, each voter will initially allocate a number of CEs (spatial redundancy) to perform redundant computation. In the execute stage, the selected CEs execute the instruction and return the results to the voters. After the execution is finished, the CEs will be released to standby. In the compare stage, the voters evaluate the returned results and determine the correctness by majority voting. If majority voting is unable to resolve the disputed results, the initially allocated CEs fail to achieve fault tolerance. The voters will store the unconfirmed results to MEMs and then allocate additional CEs to execute the same instruction (temporal and spatial redundancy) until the results eventually get confirmed. In the complete stage, incorrect results and related speculations are pruned leaving the correct results for future use. Note that to avoid resources being depleted by redundant computing, a dedicated resource management is needed to balance redundant computing and parallel processing. We will elaborate this point in section 3.

Figure 2 shows an example of the localized control in the four-stage pipeline. In the issue stage, a voter issues the instruction to its neighboring CEs. To provide fault tolerance for instruction execution, the voter will first determine the amount of redundancy (i.e., CEs) as denoted by R. Figure 2(a) shows two cases where $R = 3$ and $R = 4$, respectively. The details of the redundancy allocation policy will be discussed in section 3. To improve the efficiency, the voter will first try to allocate redundancy using the nearest CEs, as shown in the first case of Fig. 2(a). If failed, the voter will collaborate with other voters to collect enough CEs, as shown in the second case of Fig. 2(a). The latency of voter bypass is considered as a timing component in the issue stage. If more CEs need to be collected that might cause long and nondeterministic latency, the voter will wait for the neighboring CEs to become available after they finish the execute stage (which may take additional

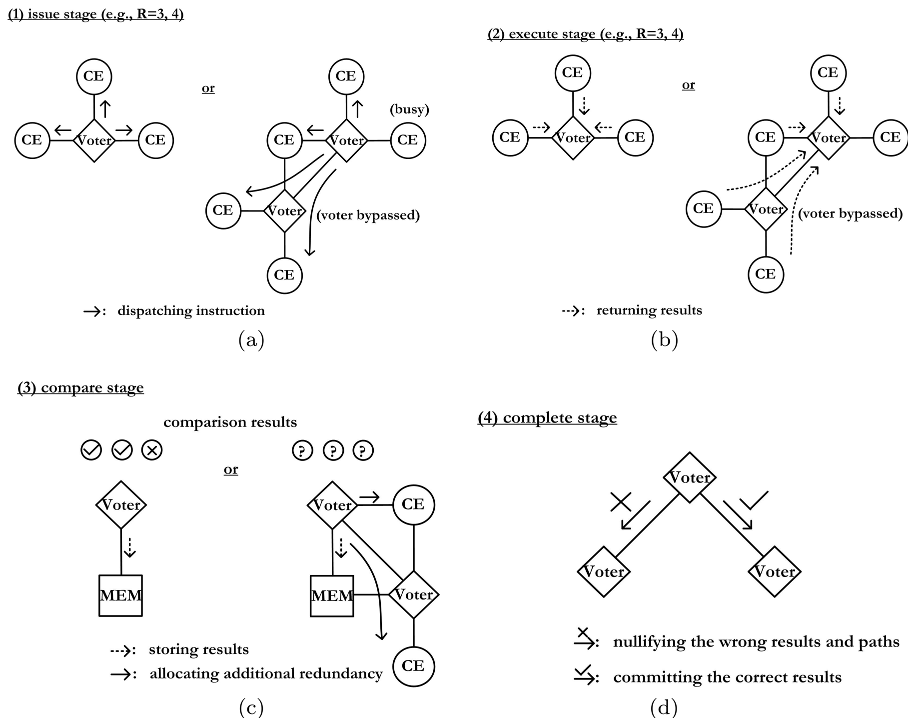

Fig. 2. An example of localized pipeline control: (a) issue stage, (b) execute stage, (c) compare stage, and (d) complete stage.

1–2 cycles). This is a type of structural hazard. In the execute stage, CEs finish the execution and return the results to the voter in charge of this instruction (see Fig. 2(b)). Next, in the compare stage, the comparison logic determines whether the results are correct or not. Meanwhile, the results are stored temporarily in the MEM for future reference. It is possible that the results cannot achieve agreement due to the faults in the execution (as the second case shown in Fig. 2(c)). Thus, the completion for this instruction has to be postponed and the voter will need to allocate additional redundancy to recover the faults. Eventually, when the execution results are confirmed, the voter will process in the complete stage (as shown in Fig. 2(d)). The incorrect results and the associated speculative execution paths are nullified and pruned by the voter, while the correct results are propagated by the voter. For other types of faults with relatively deterministic nature (e.g., the clustered faults), the affected voters may be masked out to minimize the performance degradation.

As the instruction execution is fault-prone, subsequent instructions are issued using unconfirmed results of the precedent instructions to solve data dependencies for high performance. Thus, the execution of instructions is inherently speculative. As shown in Fig. 3, the unconfirmed results of the precedent instructions propagate to provide the operands for speculative execution of the subsequent instructions (Fig. 3(a)). Again, if the voter is unable to allocate enough redundancy using

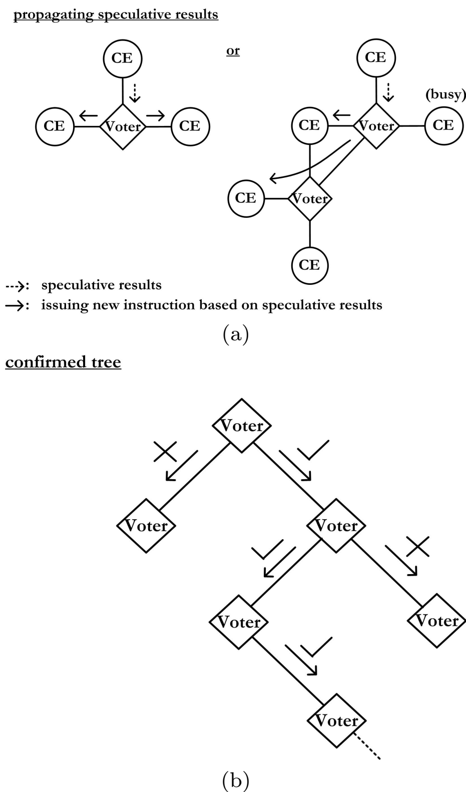

Fig. 3. Speculative execution for high performance: (a) propagating speculative results and (b) a tree formed after result confirmation.

the nearest CEs, it will cooperate with neighboring voters to issue speculative executions to other CEs, as shown in the second case in Fig. 3(a). After these results get confirmed, the wrong paths of the execution will be pruned while the right paths remain, thereby forming a tree (Fig. 3(b)) to complete the instruction execution.

3. DYNAMIC REDUNDANCY FOR RELIABLE AND SCALABLE PARALLELISM

In this section, we will apply the above nanoarchitecture model to investigate the intrinsic relationship between redundant computation and massive parallelism in nanocomputing systems. We will then develop a dynamic redundancy allocation technique that enables reliable and scalable performance in a coherent manner.

3.1 The Underlying Problem

Nanocomputing systems feature massive parallelism that can be exploited for high performance. However, massive parallelism does not necessarily lead to high performance given the high fault rates and fault recovery penalties. Although in theory fault tolerance can be achieved by redundant computation, this approach inevitably competes with parallel processing for hardware resources. How to cooperatively

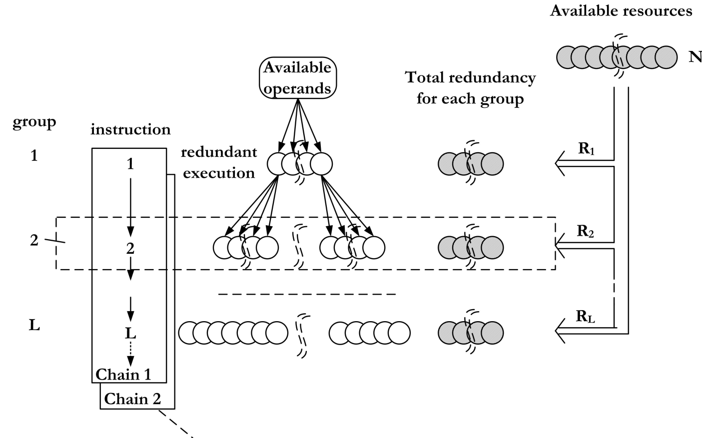

Fig. 4. The underlying problem of redundancy allocation.

manage and utilize massive parallelism for reliable and scalable performance is critical for realizing the potential of nanoscale integration.

We illustrate this problem in Fig. 4. At a given time, all the pending instructions will fall into two categories according to the dependency relations: some instructions have to be executed in series, as they closely depend on the results of their precedent instructions; and some instructions can be executed in parallel, as they are loosely coupled (e.g., they may come from different threads or become relatively independent after dependency is safely removed). Therefore, based upon these dependency relations, the instructions can be partitioned into multiple chains, where in each chain the instructions are sequentially executed (e.g., instructions 1 through L in chain 1, as shown in Fig. 4, have to be executed one by one). Meanwhile, instructions in the different chains can be executed in parallel as a group (e.g., the instruction 2 in all the chains as shown in Fig. 4 can be organized as group 2 and executed simultaneously). On one hand, it is generally desired to execute as many instructions in parallel as possible for maximal throughput. On the other hand, in nanocomputing systems each instruction requires multiple execution copies for fault tolerance. Thus, the need of providing redundancy for fault tolerance conflicts with the motivation for high performance. When the subsequent instructions in a chain are issued to avoid performance slowdown, their precedent instructions might not have got their results confirmed yet. Thus, the subsequent instructions have to be speculatively issued with multiple unconfirmed results. Consequently, how to optimally allocate the available hardware resources (the number N of CEs as shown in Fig. 4) is complex. Specifically, the hardware resources need to be allocated to each parallel executed group (e.g, R_j of CEs allocated to the jth instruction group). Meanwhile, within each group the hardware resources need to be further distributed to the individual instructions belonging to multiple chains and the redundant execution of each instruction as well.

It can be observed that the requirement for parallel processing and the need for reliable computation are competing against each other for hardware resources. When the fault rate is high, all the instructions including the precedent instructions

are difficult to complete with confirmed results. To maintain speculative execution for high parallelism, the utilization of redundancy will exponentially increase and may eventually deplete all the available resources, causing frequent structural hazards. Consequently, the course of execution will be stalled frequently because the unconfirmed instructions cannot get enough redundancy to complete the processing, and speculative execution of the subsequent instructions cannot have resources either to continue exploiting parallelism. This problem will be exacerbated if there are multiple simultaneous threads, where each group will contain many instructions due to the relatively weak dependency. Therefore, how to optimally allocate resources for reliable and scalable performance is a fundamental problem that must be addressed in nanocomputing systems. Furthermore, it is also critical to adjust the redundancy allocation at runtime according to different requirements on reliability and performance, which may vary during the course of program execution.

3.2 Parallelism Level and the Implications to Reliable Performance

As shown in Fig. 4, there are two ways to improve performance: one is to break the instructions into more chains so that more instructions can be simultaneously processed in each group; the other is to execute more groups concurrently while they are possibly in different pipeline stages. The first approach is largely determined by the nature of dependency inherent in specific programs, which is beyond control of nanocomputing architecture design. On the other hand, the number of groups executed in parallel can be increased by performing speculative execution. In this paper, we define the *parallelism level* as the number of instruction groups that can be processed concurrently. Note that the parallelism level is defined for an entire cluster, not just for a single voter or CE. As an example, a conservative design may execute an instruction only when all the dependent instructions are completed and confirmed with no faults. The instruction level in this case is 1. Another example is an aggressive design, where the system may speculatively issue and execute instructions even when the operands from precedent instructions are still not confirmed yet. As a result, there are more instruction groups being processed at the same time (possibly in different pipeline stages), which increases the parallelism level (denoted as L as shown in Fig. 4). In fault-free systems, the relationship between parallelism level and performance is straightforward: a higher parallelism level leads to better performance. In fault-prone systems, the complete stage in the pipeline may take an unpredictable number of cycles, depending on how quickly the faulty results can be recovered. This complicates the relationship between parallelism level and performance. It is likely that, while many instructions can be executed in parallel, they cannot obtain enough hardware resources to confirm their results quickly via redundant execution. As a result, instructions are frequently stalled in the pipeline, leading to a superficially "high" parallelism level which does not deliver high performance. Hence, the actual parallelism level needs to be determined carefully.

We now examine the relationship between the parallelism level and reliable performance. We assume that there are totally M instructions in the program, which can be partitioned into multiple chains and organized into totally K groups. Assume that the parallelism level is always L as shown in Fig. 4, which implies that the cluster can always process L groups of instructions simultaneously. The pipeline depth

is assumed to be D and the pipelined execution may be performed distributively on multiple CEs. The total time T for processing the M instructions specifies the execution performance. It consists of two components: regular execution time T_e (the time spent on the regular processing in all the pipeline stages, excluding stalls due to faults) and fault recovery penalty T_f (the time spent on the re-execution when uncorrectable faults are detected), expressed as

$$T = T_e + T_f. \tag{1}$$

In the regular execution time T_e, D cycles are needed for the very first group of instructions to go through the pipeline and complete. As the parallelism level is assumed to be L, the pipeline can only filled to the extent of $\frac{L}{D}$. After the pipeline is initially filled by the first L instructions, ideally the cluster can start processing the following groups one after another. This procedure only needs $K - 1$ cycles if the parallelism level is equal to the pipeline depth (i.e., $L = D$). However, if $L < D$, there are always some "bubbles" in the pipeline (which can be considered as having $(\frac{D}{L} - 1)$ NOP groups inserted immediately after each instruction group without delivering any performance). Thus, in fault-free systems, the parallelism level is upper-bounded by the pipeline depth, i.e., $L \leq D$. The fault-free systems can achieve the optimal performance when $L = D$. On the other hand, in fault-prone systems, the parallelism level L is no longer bounded by the pipeline depth D. If the faulty results cannot be quickly recovered, the compare stage may take an unpredictable number of cycles to finish. As a result, there might exist $L > D$ instruction groups being processed concurrently. However, this does not deliver high performance as the regular execution time T_e will not be reduced. In fact, this case indicates the existence of stalls, and hence the non-zero fault recovery penalties T_f, as will be discussed later. The regular execution time T_e can be derived as

$$T_e = \begin{cases} D + (K-1)\frac{D}{L}, & 0 < L \leq D, \\ D + K - 1, & L > D. \end{cases} \tag{2}$$

From (2), the optimal parallelism level should be $L^{opt} = D$ for both fault-free and fault-prone systems. Note that in ideal fault-free systems the actual parallelism level is always smaller than L^{opt}. This is because not all the opportunities of parallel execution can be detected and exploited in practice, especially considering pipeline bubbles due to control and structural hazards. When L is much less than L^{opt}, the utilization of parallelism tends to be over-conservative. As a result, execution takes a longer time according to (2), leading to performance slowdown even when there is no penalty T_f. On the other hand, faults in real systems may drive the parallelism level away from the optimal L^{opt}. Once uncorrectable faults occur, the comparison fails to resolve the disputed results, thereby the system has to allocate additional redundancy (temporal and/or spatial) for re-execution. Hence, stalls occur frequently that hold more instructions than expected. This may increase L beyond L^{opt}, especially if the parallelism is over-aggressively utilized. However, this increase in L only leads to performance degradation. Thus, both over-conservative and over-aggressive utilization of parallelism will cause performance slowdown. An optimal nanoarchitecture solution should aim at closing the gap between L and L^{opt} as much as possible.

3.3 Scalable Parallelism via Dynamic Redundancy Allocation

We now present the dynamic redundancy allocation technique that optimizes the resource management under high fault rates, thereby enabling scalable parallelism for jointly achieving reliable and high-performance nanocomputing.

Assume that the redundancy allocated for parallel execution of the instructions in group j is R_j of CEs and the total number of available CEs is N. The central idea of our approach is to adjust R_j dynamically so that the parallelism level L is maintained at L^{opt}. There is a constraint regarding the availability of redundancy, expressed as follows

$$\sum_{j=1}^{L} R_j \leq N. \tag{3}$$

If the requests for redundancy are more than that available, unsatisfied requests combined with new requests during the following cycles will be accumulated quickly and eventually push the parallelism away from the optimal level. Simulation results in section 4 clearly show that techniques unaware of (3) lead to performance degradation.

Consider the fact that the available redundancy N may vary over time due to the runtime variations of instruction processing. Instead of trying to find the specific amount of redundancy allocated to each instruction, we focus on the relative ratios that are more stable for allocating the available redundancy. Thus, the constraint (3) can be recast as

$$\sum_{j=1}^{L} \alpha_j \leq 1, \tag{4}$$

where $\alpha_j = \frac{R_j}{N}$ defines the ratio between R_j and N, both of which are time-varying in general.

As discussed before, when the actual parallelism level is greater than the optimal L^{opt}, stalls due to uncorrectable faults occur, which hurts the overall performance. Since speculative execution based on the faulty results of the precedent instructions has already consumed too many resources at this time, it does not make sense to continue allocating redundancy for further speculation. Therefore, we only need to allocate redundancy for the instructions in groups from 1 up to L^{opt} at runtime, i.e.,

$$\alpha_j = \begin{cases} \alpha_j^{opt}, & \text{if } 1 \leq j \leq L^{opt}, \\ 0, & \text{if } j > L^{opt}. \end{cases} \tag{5}$$

where $\{\alpha_j^{opt}\}_{j=1}^{L^{opt}}$ are the optimal ratios of assigning R_j of CEs to the instructions in group j and $\sum_{j=1}^{L^{opt}} \alpha_j^{opt} = 1$ so that (4) can be always satisfied. Doing so avoids over-aggressive speculative processing that may potentially deplete the resources.

In order to determine the optimal ratios α_j^{opt}'s, we consider the fault recovery penalty T_f in (1). It is determined by the number of re-executions as well as the penalty associated with these re-executions. Note that the instructions have different probabilities of being re-executed. Thus, we consider the mean value t_f,

which is the statistical measure of T_f, expressed for group 1 through L as

$$t_f = \sum_{j=1}^{L} P_j \overline{\delta}, \tag{6}$$

where P_j is the probability of re-execution of instructions in group j and $\overline{\delta}$ is the average penalty of each re-execution.

To calculate the re-execution probability P_j, we need to consider the probability of having uncorrectable faults. Assume that due to faults a CE will generate wrong results with a probability f, referred to as the fault rate of the CEs. If an instruction is executed with redundancy R, faults cannot be resolved when the number of correct results is less than two, as the majority voting cannot select the correct results out of the incorrect ones. Note that the majority voting may make wrong decisions under certain fault patterns. This is a limitation of majority voting, which will affect all fault-tolerant techniques using majority voting or similar schemes for fault detection. However, fault tolerance can be achieved at a confident level when assigning enough amount of redundancy R for a given fault rate f. The probability of having uncorrectable faults due to lack of correct results is given by $q(f, R)$ as follows

$$q(f, R) = f^R + R(1 - f)f^{R-1}. \tag{7}$$

If the redundancy for the instructions in group 1 is R_1, these instructions will have a re-execution probability $P_1 = q(f, R_1)$. Since the instructions in group 2 with redundancy R_2 are executed speculatively when their precedent instructions in group 1 have not been confirmed yet, each of the R_1 copies of instructions in group 1 is assigned with $\frac{R_2}{R_1}$ redundancy on average (assuming R_1 is a factor of R_2) when executing the corresponding instructions in group 2. The re-execution probability of the instructions in group 2 is thus expressed as $P_2 = P_1 + (1 - P_1)q(f, \frac{R_2}{R_1})$. This is obtained based on the following observations. If the instructions in group 1 need to be re-executed (with probability P_1), the corresponding instructions in group 2, which are speculatively executed based on the instructions in group 1, should definitely be re-executed as well. On the other hand, even if the instructions in group 1 do not need re-execution (with probability $1 - P_1$), the instructions in group 2 may still have to be re-executed (with probability $q(f, \frac{R_2}{R_1})$) due to their faulty CEs. Proceeding in the same fashion, we obtain the re-execution probability for the instructions in group j as

$$P_j = P_{j-1} + (1 - P_{j-1})q(f, \frac{R_j}{R_{j-1}}), \tag{8}$$

where $j > 1$ and $P_1 = q(f, R_1)$.

From (7) and (8), the re-execution probability of all the instructions can be calculated in a recursive way. Thus, the fault recovery penalty t_f can be evaluated according to (6). Since α_j's determine the redundancy allocation, the optimal ratios

α_j^{opt}'s should satisfy

$$\text{minimize:} \sum_{j=1}^{L^{opt}} P_j, \tag{9}$$

$$\text{subject to:} \sum_{j=1}^{L^{opt}} \alpha_j = 1. \tag{10}$$

It is in general very difficult to analytically determine the values of α_j^{opt}'s. However, the following guidelines can be applied to estimate α_j^{opt}'s.

(i) From (7), fault tolerance is possible if $R > 1$, which is very much in line with the well-prevalent notion of redundancy-based execution. Thus, $\frac{R_j}{R_{j-1}}$ in (8) should be greater than 1, implying $\alpha_j > \alpha_{j-1}$. Thus, we should increase the values of α_j^{opt}'s from the precedent instructions to the subsequent instructions.

(ii) From (8), the re-execution probability (and hence the recovery penalty) are accumulated from the precedent instructions to the subsequent instructions. This implies that quick confirmation of the precedent instructions is important. Given this observation, more resources should be made available for fault tolerance in each copy of the precedent instructions. According to (8), a practical way to achieve this is to make $\frac{R_{j+1}}{R_j} < \frac{R_j}{R_{j-1}}$. This requires $\frac{\alpha_{j+1}^{opt}}{\alpha_j^{opt}} < \frac{\alpha_j^{opt}}{\alpha_{j-1}^{opt}}$, where the increase in the ratios α_j^{opt}'s needs to be slowed down from the precedent instructions to the subsequent instructions.

By minimizing the fault recovery penalty, the proposed dynamic redundancy allocation technique is able to improve not only the performance but also the predictability of performance. This will be shown in section 4.

Note that the above analysis can be easily extended to the general case where $\frac{R_j}{R_{j-1}}$ may not be an integer. Assume that $\frac{R_j}{R_{j-1}} = r + \Delta r$, where we define $r = \lfloor \frac{R_j}{R_{j-1}} \rfloor$ and $\Delta r = \frac{R_j}{R_{j-1}} - r$. When the average redundancy for each speculative execution is $\frac{R_j}{R_{j-1}}$ (where R_{j-1} is the total redundancy of the precedent group of instructions), we should allocate r and $r+1$ redundancy for speculative executions based on the $R_{j-1}(1-\Delta r)$ and $R_{j-1}\Delta r$ unconfirmed results, respectively, of the precedent instructions. Therefore, (7) can be extended to

$$q(f,R) = \begin{cases} f^R + R(1-f)f^{R-1}, & R \in \mathbf{Z}, \\ (1-\Delta r)q(f,r) + \Delta r \; q(f,r+1), & \text{otherwise}, \end{cases} \tag{11}$$

where $r = \lfloor R \rfloor$ and $\Delta r = R - r$.

We use an example to show the selection of the optimal allocation ratios. In this example, $L^{opt} = D = 4$ is assumed. The total number of available CEs is $N = 20$ and $N = 40$ for demonstration purpose. We vary the fault rate in the range of $f \in [0.1, 0.5]$ and search for α_j^{opt}'s to minimize the fault recovery penalty. The search is done in the space where α_j^{opt}'s are rounded to the closest $1/N$ so that $N\alpha_j^{opt}$'s are integers. As shown in Table I, when N increases, for the same f we should allocate relatively more redundancy for the subsequent instructions. This is because the precedent instructions can obtain enough redundancy for quick

Table I. An example of selecting optimal redundancy allocation ratios.

N	f	α_1^{opt}	α_2^{opt}	α_3^{opt}	α_4^{opt}
20	0.1	0.10	0.20	0.30	0.40
20	0.2	0.15	0.25	0.25	0.35
20	0.3	0.15	0.25	0.25	0.35
20	0.4	0.20	0.25	0.25	0.30
20	0.5	0.20	0.25	0.25	0.30
40	0.1	0.075	0.150	0.300	0.475
40	0.2	0.075	0.200	0.350	0.375
40	0.3	0.100	0.275	0.300	0.325
40	0.4	0.100	0.275	0.300	0.325
40	0.5	0.125	0.275	0.275	0.325

confirmation even with the smaller ratios. If f increases but N is fixed, more redundancy should be allocated to the precedent instructions to minimize the re-execution probabilities.

Note that the α_j^{opt}'s can be determined in the design phase based on availability of N and estimated f from the physical systems. These ratios can be employed thereafter to guide the dynamic redundancy allocation at runtime.

3.4 Dynamic Redundancy Allocation Algorithm

The general procedure of dynamic redundancy allocation is summarized in Algorithm 1. Assume that the instruction being processed is in group j, and there are totally k voters in this group. The goal of this algorithm is to determine when to allocate redundancy and how much redundancy should be allocated.

Algorithm 1: The general procedure of dynamic redundancy allocation.

```
Input:
status_parent (status of the parent voter),
confirmation (confirmation status of the current voter),
N (total availability),
j (group index),
k (number of voters simultaneously processing in the same group)
begin
    while the voter is active do
        if status_parent is Invalid then
            nullify all the execution under control of this voter
            propagate info. to the subsequent voters
        end
        else if confirmation is False then
            R = calculate_redundancy(N, j, k)
            allocate R redundant computation from neighboring CEs
        end
    end
end
```

Algorithm 2: *calculate_redundancy*(N, j, k).

Input: N, j, k
Output: R
begin
  if j *is changed* **then**
    read α_j^{opt} from LUT indexed by j
  if N *is changed* **then**
    LUT-based calculation for $N\alpha_j^{opt}$
  LUT-based calculation for $R = N\alpha_j^{opt}\frac{1}{k}$
  return R
end

If the parent voters notify the current voter that the operands the current voter is using have been proved to be invalid, the current voter should nullify all the involved results and propagate the information to the involved voters. Otherwise, the current voter needs to check whether the execution has achieved majority agreement. If the results are not confirmed yet, the current voter will try to allocate more redundancy.

Next, the amount of redundancy is determined according to the proposed dynamic redundancy allocation strategy. This function is generalized in Algorithm 2. It is essentially performing $\frac{N\alpha_j^{opt}}{k}$. By doing so, the redundancy can be distributed evenly to each instructions in this group for the sake of fairness. Note that prioritized allocation in favor of specific instructions is also feasible. The complexity of redundancy allocation consists mainly of LUT, as the voters just perform LUT-based operations to allocate redundancy as shown in Algorithm 1 and Algorithm 2. Furthermore, some operations shown in Algorithm 2 do not need to be performed every cycle. For example, the operation for getting α_j^{opt} is not necessary if the group number j remains the same. Thus, the algorithm can come up with a timely decision. In addition, the timing efficiency can be further improved by separating the issue stage into two stages.

4. EVALUATION AND DISCUSSION

In this section, we evaluate the proposed dynamic redundancy allocation technique. We first explain the evaluation methodology and then discuss the simulation results.

4.1 Methodology

The simulations are implemented in C program based on the proposed dynamic redundancy allocation technique. Here we are interested in architecture-level abstracted behaviors instead of implementation or program details. To evaluate the effectiveness of the proposed technique, we compare our work with two existing techniques: one uses fixed dual-modular redundancy, where speculative execution always takes two computation units; and the other is an improved technique [Rao et al. 2005; Rao et al. 2007], which assigns either one processing unit if the previous instruction can still be confirmed, or two if the previous instruction is not confirmable. In order to ensure a fair comparison, all the settings are equivalent

to the two existing techniques except for the redundancy allocation algorithm. In particular, we compare the performance of our technique with the two existing techniques under the same number of CEs. We also make the same assumption on the utilization of MEMs. The focus of this work is on how to efficiently utilize available computational hardware resources (CEs) to jointly achieve fault tolerance and high performance under a range of fault rates. In the simulations, all the instructions are dependent on their immediate precedent instructions. In such an example, instructions are speculatively executed based on unconfirmed results. Other programs of different dependencies can be considered as a combination of multiple copies of this specific example but with different numbers of instructions and appearing at different phases in the program. There are totally 40 CE's in the cluster and $L^{opt} = D = 4$. The optimal ratios α_j^{opt}'s for dynamic redundancy allocation are selected according to the method described in section 3.

In these simulations, we deliberately introduce faults into the execution results of CEs and evaluate the performance in terms of cycle per instruction (CPI). These faults represent the possible upsets during CE execution as well as in the associated interconnects and MEM units. We evaluate our technique using abstracted instructions, which do not depend on specific applications but do represent many key computational activities in real tasks. Note that only the correctness of execution, rather than the actual execution results, matters to this evaluation. Thus, abstracted instructions not targeting program details are sufficient for this study. The voters are assumed to be reliable as explained in section 2. Each time when a CE is processing an instruction, the result may go wrong at a certain rate, which is contributed by both permanent defects and transient errors. Without a general fault model for nanoscale systems, we may assume for the purpose of demonstration that faults occur independently. Note that the fault rate of CEs is generally higher than the fault rate of individual nanoelectronic devices. Thus, a large range of fault rates are evaluated for the proposed technique to account for the effect of increase in failure probability at the coarse granularity.

4.2 Average Performance

Figure 5 compares the average CPI for fault rates ranging from 0 to 0.5. The fault rates here are the possibility that the execution results of CEs go wrong. As we consider in-order execution, the minimum CPI is close to 1. Employing the proposed technique, the average CPI can approach this bound even at rather high fault rates. In addition, our technique achieves approximately $20\% - 30\%$ improvement on the average CPI as compared to the two fixed redundancy allocation techniques.

An interesting result is that when the fault rate increases, the proposed dynamic redundancy technique maintains a very stable performance as compared with the other two techniques. Using the two fixed redundancy techniques, the average CPI first reduces then increases as the fault rate increases. This is because the fixed redundancy techniques utilize redundancy rigidly neglecting the changing demands of fault tolerance and high performance. When the fault rate is low, the execution results are easily confirmed. However, the fixed redundancy schemes still utilize the same amount of redundancy for fault tolerance, thereby wasting resources that could otherwise be used for enhancing the performance. As the fault rate starts to increase, some speculative executions are corrupted and the incorrect paths are

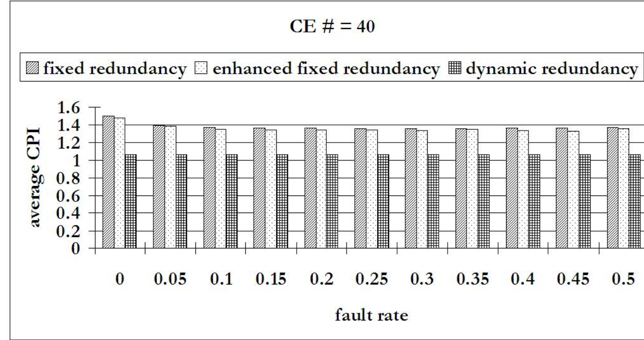

Fig. 5. Comparison of average CPI.

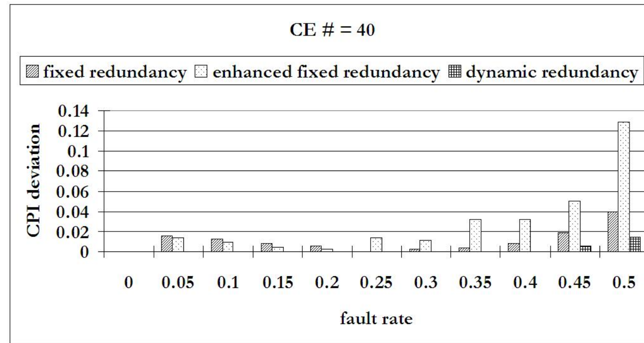

Fig. 6. Comparison of CPI deviation.

pruned. This effectively releases some resources for performance improvement. As a result, we observe a reduction in average CPI when the fault rate increases from 0 to around 0.25 in fixed redundancy allocation schemes. When the fault rate continues to increase, the fixed redundancy techniques show disadvantages manifested as the performance slowdown. While the executions are corrupted at a higher rate, these schemes only provide a fixed amount of redundancy for each allocation, which is unlikely to be sufficient to confirm the instructions quickly. As a result, although some efforts can be saved from the pruned incorrect branches, the overall performance can hardly be satisfactory due to the lagging instruction confirmation and frequent re-execution for fault recovery.

4.3 Performance Predictability

Performance predictability is an important metric, especially for real-time applications which typically require stable and predictable system performance. Similar to the simulations described above, we study the standard deviation of CPI under different fault rates. The results are shown in Fig. 6. As indicated, the performance of the fixed redundancy techniques shows large variations. When the fault rate is around 0.5, the CPI deviation is more than 12%. In contrast, the proposed dynamic redundancy technique significantly reduces the CPI deviation, e.g., only at 1.5%

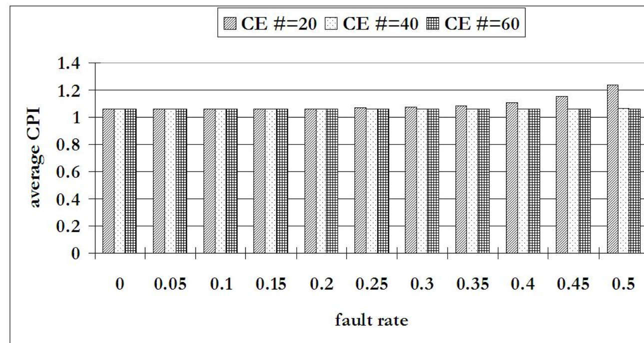

Fig. 7. The impact of available CEs on average CPI.

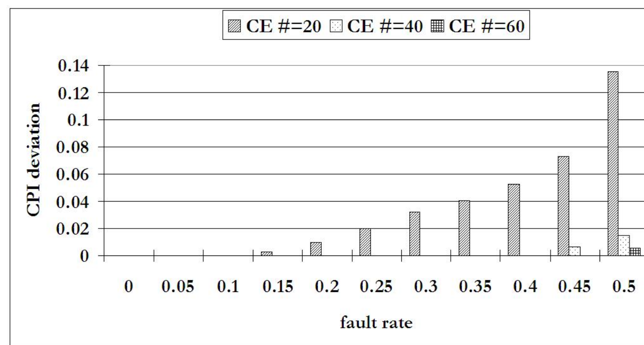

Fig. 8. The impact of available CEs on CPI deviation.

when the fault rate is 0.5, and almost zero when the fault rate is smaller.

It can be observed that the proposed technique is able to deliver stable performance over a range of fault rates. The reason is that dynamic redundancy allocation is flexible and can allocate more redundancy to the precedent instructions, thereby not only preventing resources from being depleted by over-aggressive speculative execution but also accelerating the confirmation of precedent instructions. This is important to reduce re-execution, especially when the fault rate is high. Therefore, the proposed technique enables high performance with a predictable manner. This is also beneficial for synchronizing with other nanocomputing subsystems.

4.4 Scalability

We also study the scalability of the proposed technique. Figures 7 and 8 show the results of average CPI and the deviations with different numbers of CEs under a range of fault rates. We can easily see that the average performance of the proposed technique is very stable for different numbers of CEs when the fault rate is below 0.3. When the system is experiencing a rather high fault rate at around 0.5, the dynamic redundancy allocation can still achieve an average CPI at around 1.2 with only 20 CEs. This result is even better than the fixed redundancy techniques with 40 CEs (see Fig. 5). Thus, the proposed technique can reduce the hardware

resources needed for fault tolerance and performance. In addition, the dynamic redundancy allocation maintains performance predictability at different numbers of CEs, as shown in Fig. 8. Thus, the results in Fig. 7 and 8 justify the good scalability of the proposed technique.

5. CONCLUSION

In this paper, we propose a reliable and high-performance nanoarchitecture solution targeting the emerging challenges in nanocomputing systems. A dynamic redundancy allocation strategy is developed to manage the parallelism to an optimal level so that both fault tolerance and high performance can be jointly achieved. Simulation results demonstrate performance improvement over the existing redundancy management techniques under a range of fault rates. In addition, the performance improvement is quite stable and scalable under different operating conditions. Our future work is directed to architecture-level solutions for communication and close collaboration between nanocomputing elements.

Acknowledgment

This research was supported by the NSF grant CCF-0621947 and the University of Connecticut Faculty Research Grant 446751.

REFERENCES

Martel, R., Derycke, V., Appenzeller, J., Wind, S., and Avouris, P. 2002. "Carbon nanotube field-effect transistors and logic circuits," *ACM IEEE Design Automation Conference*, 94–98.

Huang, Y., Duan, X., Cui, Y., Lauhon, L. J., Kim, K-Y., and Lieber, C. M. 2001. "Logic gates and computation from assembled nanowire building blocks," *Science*, *294,* 5545, 1313–1317.

Lent, C. S., Tougaw, P. D., Porod, W., and Bernstein, G. H. 1993. "Quantum cellular automata," *Nanotechnology*, *4,* 4, 49–57.

Ma, X., Huang, J., and Lombardi, F. 2008. "A model for computing and energy dissipation of molecular QCA devices and circuits," *ACM Journal on Emerging Technologies in Computing Systems*, *3,* 4, 18:1–18:30.

Mazumder, P., Kulkarni, S., Bhattacharya, M., Sun J. P., and Haddad, G. I. 1998. "Digital circuit applications of resonant tunneling devices," *Proceedings of the IEEE*, *86,* 4, 664–686.

DeHon, A. 2005. "Nanowire-based programmable architecture," *ACM Journal on Emerging Technologies in Computing Systems*, *1,* 2, 109–162.

Wang, T., Qi, Z., and Moritz, C. A. 2004. "Opportunities and challenges in application-tuned circutis and architectures based on nanodevices," *First ACM Conference on Computing Frontier*, 503–511.

Koren, I., Koren, Z., and Stapper, C. H. 1994. "A statistical study of defect maps of large area VLSI IC's," *IEEE Transactions on VLSI Systems*, *2,* 2, 249–256.

Mishra, M. and Goldstein, S. C. 2003. "Defect tolerance at the end of the roadmap," *International Test Conference*, 1201–1211.

Tahoori, M. B. 2005. "A mapping algorithm for defect-tolerance of reconfigurable nano-architectures," *IEEE International Conference on Computer-Aided Design*, 668–672.

Jeffery, C. M. and Figueiredo, R. 2006. "Hierarchical fault tolerance for nanoscale memories," *IEEE Transactions on Nanotechnology*, *5,* 44, 407–414.

Kuekes, P. J., Robinett, W., Seroussi, G. and Williams, R. S. 2005. "Defect-tolerant interconnect to nanoelectronic circuits: internally redundant demultiplexers based on error-correcting codes," *Nanotechnology*, *16,* 6, 869–882.

von Neumann, J. 1956. "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in C. Shannon and J. McCarthy, editors, *Automata Studies*, Princeton University Press.

Han, J. and Jonker, P., 2002. “A system architecture solution for unreliable nanoelectronic devices,” *IEEE Transactions on Nanotechnology, 1,* 4, 201–208.

Bhaduri, D., Shukla, S., Graham, P., and Gokhale, M., 2007. “Comparing reliability-redundancy tradeoffs for two von Neumann multiplexing architectures,” *IEEE Transactions on Nanotechnology, 6,* 3, 265–279.

Roy, S. and Beiu, V., 2005. “Majority multiplexing-economical redundant fault-tolerant designs for nanoarchitectures,” *IEEE Transactions on Nanotechnology, 4,* 4, 441–451.

Rao, W., Orailoglu, A., and Karri, R. 2005. “Architecture-level fault tolerant computation in nanoelectronic processors,” *IEEE International Conference on Computer Design*, 533–542.

Rao, W., Orailoglu, A., and Karri, R. 2007. “Towards nanoelectronics processor architectures,” *Journal of Electronic Testing: Theory and Applications, 23*, 235–254.

Nicolau, A. and Fisher, J. A. 1984. “Measuring the parallelism available for very long instruction word architectures,” *IEEE Transactions on Computers, 33,* 11, 968–976.

Wall, D. W. 1993. “Limits of instruction-level parallelism,” *Wester Research Laboratory Research Report 93/6*, Digital Equipment Corporation.

Tullsen, D. M., Eggers, S. J., and Levy, H. M. 1995. “Simultaneous multithreading: maximizing on-chip parallelism,” *International Symposium on Computer Architecture*, 392–403.

Lo, J., Eggers, S., Emer, J., Levy, H., Stamm, R., and Tullsen, D. M. 1997. “Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading,” *ACM Transaction on Computer Systems, 15,* 3, 322–354.

Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., and Chang, K. 1996. “The Case for a Single Chip Multiprocessor,” *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2–11.

Hammond, L., Nayfeh, B., and Olukotun, K. 1997. “A single-chip multiprocessor,” *Computer, 30,* 9, 79–85.

Wang, S., Wang, L., and Jain, F. 2007. “Dynamic redundancy allocation for reliable and high-performance aanocomputing,” *International Symposium on Nanoscale Architectures*, 1–6.

Beckett, P. and Jennings, A. 2002. “Towards nanocomputer architecture,” *Asia-Pacific Conference on Computer Systems Architecture*, 141–150.

Fountain, T. J., Duff, M. J. B. D., Crawley, D. G., Tomlinson, C. and Moffat, C. 1998. “The use of nanoelectronic devices in highly-parallel computing systems,” *IEEE Transactions on VLSI Systems, 6,* 1, 31–38.

Franklin, M. and Sohi, G. S. 1992. “The expandable split window paradigm for exploiting fine-grain parallelism,” *International Symposium on Microarchitecture*, 58–67.

Farkas, K., Chow, P., Jouppi, N., and Vranesic, Z. 1997. “The multicluster architecture: reducing cycle time through partitioning, ” *International Symposium on Microarchitecture*, 149–159.

Goldstein, S. and Budiu, M. 2001. “NanoFabrics: spatial computing using molecular electronics,” *International Symposium on Computer Architecture*, 178–189.

Zhu, W. and Fleisch, B. 2000. “Performance evaluation of soft real-time scheduling on a multicomputer cluster,” *International Conference on Distributed Computing Systems*, 610–617.

He, L., Jarvis, S. A., Spooner, D. P., Chen, X., and Nudd, G. R. 2004. “Dynamic scheduling of parallel jos with QoS demands in multiclusters and grids,” *International Workshop on Grid Computing*, 402–409.

Tang, X. Y. and Chanson, S. T. 2001. “Optimizing static job scheduling in a network of heterogeneous computers,” *International Conference on Parallel Processing*, 373–382.

Chen, Y., Jung, G.-Y., Ohlberg, D. A. A., Li, X., Stewart, D. R., Jeppesen, J. O., Nielsen, K. A., Stoddart, J. F., and Williams, R. S. 2003. “Nanoscale molecular-switch crossbar circuits,” *Nanotechnology, 14,* 4, 462–468.