A Hybrid Nano-CMOS Architecture for Defect and Fault Tolerance

MUZAFFER O. SIMSIR
Princeton University
SRIHARI CADAMBI, FRANJO IVANČIĆ, and MARTIN ROETTELER
NEC Laboratories America
and
NIRAJ K. JHA
Princeton University

14

As the end of the semiconductor roadmap for CMOS approaches, architectures based on nanoscale molecular devices are attracting attention. Among several alternatives, silicon nanowires and carbon nanotubes are the two most promising nanotechnologies according to the ITRS. These technologies may enable scaling deep into the nanometer regime. However, they suffer from very defect-prone manufacturing processes. Although the reconfigurability property of the nanoscale devices can be used to tolerate high defect rates, it may not be possible to locate all defects. With very high device densities, testing each component may not be possible because of time or technology restrictions. This points to a scenario in which even though the devices are tested, the tests are not very comprehensive at locating defects, and hence the shipped chips are still defective. Moreover, the devices in the nanometer range will be susceptible to transient faults which can produce arbitrary soft errors. Despite these drawbacks, it is possible to make nanoscale architectures practical and realistic by introducing defect and fault tolerance. In this article, we propose and evaluate a hybrid nanowire-CMOS architecture that addresses all three problems—namely high defect rates, unlocated defects, and transient faults—at the same time. This goal is achieved by using multiple levels of redundancy and majority voters. A key aspect of the architecture is that it contains a judicious balance of both nanoscale and traditional CMOS components. A companion to the architecture is a compiler with heuristics to quickly determine if logic can be mapped onto partially defective nanoscale elements. The heuristics make it possible to introduce defect-awareness in placement and routing. The architecture and compiler are evaluated by applying the complete design flow to several benchmarks.

This article is an extended version of the paper "Fault-tolerant computing using a hybrid nano-CMOS architecture," presented at the International Conference on VLSI Design, 2008.
Authors' addresses: M. O. Simsir (corresponding author), Department of Electrical Engineering, Princeton University, Princeton, NJ 08544; S. Cadambi, F. Ivančić, M. Roetteler, NEC Laboratories America, Princeton, NJ 08540; N. K. Jha, Department of Electrical Engineering, Princeton University, Princeton, NJ 08544.

DOI 10.1145/1568485.1568488 http://doi.acm.org/10.1145/1568485.1568488

Categories and Subject Descriptors: B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms: Design, Reliability

Additional Key Words and Phrases: Defect tolerance, nanotechnology, nanowires

ACM Reference Format:

Simsir, M. O., Cadambi, S., Ivančić, F., Roetteler, M., and N. K. Jha. 2009. A hybrid nano-CMOS architecture for defect and fault tolerance. ACM J. Emerg. Technol. Comput. Syst. 5, 3, Article 14 (August 2009), 26 pages.

DOI = 10.1145/1568485.1568488 http://doi.acm.org/10.1145/1568485.1568488

1. INTRODUCTION

As we come closer to the end of the semiconductor roadmap for CMOS technology, nanoscale molecular devices constructed from silicon nanowires and carbon nanotubes are emerging as alternatives. ITRS [ITRS 2005] rates these devices as among the most promising of all the new technologies under investigation.

Nanowire and carbon nanotube devices that implement logic functions, interconnect, and memory elements have unmatched densities compared to their CMOS counterparts. However, this advantage does not come for free. A major drawback is their defect-prone manufacturing process. The defect rate is expected to be as high as 10% [DeHon 2005]. However, since these devices have extremely high densities, which lead to low controllability and observability, comprehensive defect location is very difficult. Most testing methods proposed in the literature [Mishra and Goldstein 2003; DeHon and Naeimi 2005; Brown and Blanton 2004; Jacome et al. 2004] are probabilistic; that is, they isolate regions of the chip in which a fault likely exists, but do not identify or localize faults with certainty. It is unlikely that such methods will scale to be able to test a large chip with reasonable speed and accuracy. The complexity of testing, coupled with high defect rates, imposes a constraint on the architecture; it must be defect-tolerant. This observation is essential to make nanoscale molecular devices practical.

The nanoscale devices fabricated by using nanowire technology have the reconfigurability property which permits defect tolerance. For example, diode-based logic [DeHon 2005], which uses intersecting horizontal and vertical nanowires, uses the reconfiguration property to implement logic. The junction of nanowires is configured by applying a voltage. The configuration does not change if the circuit is operated at lower voltages. This also implies that each junction actually stores a configuration bit. However, it is possible to reconfigure the chip by applying high voltages again, implying that the same chip can be used to implement other logic functions later on. This also means that the defective components can be avoided by "compiling around" the defects, that is, the compiler can choose not to use defective components. Nevertheless, this approach can only be followed for the defects that are located during testing. Therefore, defect tolerance by reconfigurability is not a complete solution, given that testing methods cannot locate all defects.

Another problem associated with nanoscale molecular devices are transient faults, which are temporary glitches that cause erroneous behavior. Transient faults can occur due to cosmic rays, heat and noise [Snider et al. 2005; Shivakumar et al. 2002; Rejimon and Bhanja 2006; Mukherjee et al. 2005]. Small structures, such as nanoscale circuits, are particularly susceptible to them. Therefore, even if all defects can be located and avoided during implementation, transient faults can still cause the circuit to malfunction for a short period of time.

Although methods to tolerate fabrication defects have been addressed in the nanotechnology community, the problems of unlocated defects and transient faults are not well investigated yet. In this article, we propose an architecture that attempts to provide one solution to all these problems at the same time. An accompanying compiler is also presented that allows us to evaluate the efficacy of the architecture.

Our work is characterized by the following key contributions:

—We propose an architecture with two levels of redundancy: first, each processing element is based on nanowire crossbars that contain extra nanowires, and second, the processing elements themselves are replicated.
—We "hybridize" the architecture by using CMOS logic to vote on the outputs of the redundant nanoelectronic processing elements. The redundancy and voting mechanisms resolve transient faults as well as unlocated permanent faults.
—We present the effect of using CMOS logic for voting on area overhead and transient fault tolerance by comparing to its nanowire counterpart.
—We propose fast compilation heuristics so that compilation time is largely unaffected by the defect rate.
—Our compiler has distinct defect-aware and defect-unaware phases, and is retargetable across defect models. A key idea that helps achieve this is that all defect information is completely subsumed within the cost function of the simulated annealing-based placer.
—The proposed architecture is evaluated by using different manufacturing defect probabilities and various nanowire processing elements with different complexities.

The rest of the article is organized as follows. In Section 2, we provide background information about nanowire-based design and the CMOS-nanowire interface. In Section 3, we present our proposed hybrid fault-tolerant architecture and discuss various architectural parameters. In Section 4, we present the compilation framework and algorithms. In Section 5, we present experimental results and conclude in Section 6.

2. BACKGROUND

This section presents prior work on nanowire-based architectures and CMOS-nanowire interfaces.

2.1 Nanowire-Based Architectures

Nanowires are wires that have diameters on the order of nanometers and lengths up to several hundreds of micrometers. By using metallic, p-type, and n-type nanowires, basic computation blocks, such as AND, OR and NOR gates [Huang et al. 2001], can be implemented. Nanowires can also be used for nonvolatile storage using bistable nanoscale switches [Duan et al. 2002].

Furthermore, nanowires can be aligned to construct the so-called crossbar architectures, in which a layer of nanowires is placed orthogonal to another layer. In such architectures, the region where two wires in different layers overlap is called a junction. When activated by a voltage, each junction behaves like a diode, field-effect transistor (FET), or resistor, depending on the electrical properties of the nanowires and the layer between them. A function can be realized by configuring selected junctions in the crossbar.

Among all nanowire architectures, crossbar architectures are the most dominant since nanoscale manufacturing processes yield regular structures which make them easier to build. These architectures also permit easy incorporation of redundancy, which makes them suitable in the presence of high manufacturing defect rates. The ability to configure each junction individually allows routing around the defective junctions. Nondefective junctions can be used for computation.

A crossbar architecture proposed in DeHon [2005] uses programmable diode junctions to construct OR planes, and nanowire FET-based buffers and inverters to make up NOR and OR planes. Another crossbar architecture from HP Labs [Snider et al. 2005] has junctions that can be configured as FETs or diodes, depending on the material between the nanowire layers. n-type and p-type FETs are used to implement logic, while diode junctions are used for routing. In these two proposals, known defects are avoided by routing around them. Another architecture, which is a CMOS-nanowire hybrid chip, proposed in Rad and Tehranipoor [2006], uses an island-style FPGA architecture. The logic blocks are implemented using nanowires, and the interconnect using CMOS. However, it is assumed that nanowire-based logic blocks are defect-free.

In general, existing architectural proposals tolerate manufacturing defects using configurability and redundancy. However, this requires that the defects must be identified and localized, which may not be feasible for nanoscale architectures. Also, previous work has not addressed the possibility of frequently occurring transient faults.

2.2 CMOS-Nanowire Interface

In order to implement logic on nanowire crossbars, each nanowire must be accessed individually to be able to program selected junctions. Since it is not possible to have direct access to nanoscale wires, a CMOS-nanowire interface should be provided. Furthermore, this interface can not only be used to program junctions, but also serve as a signal input/output interface between nanowire and CMOS components for hybrid designs.

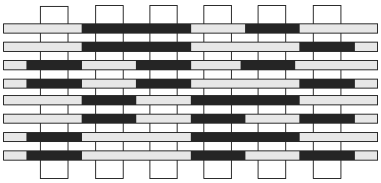


Fig. 1. CMOS-nanowire interface using a nanowire decoder.

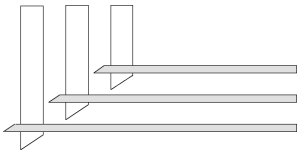


Fig. 2. CMOS-nanowire interface using one-to-one mapping.

2.2.1 *Implementation Details.* In the literature, there are three main CMOS-nanowire interface proposals. In the first proposal, a nanowire decoder is used to activate exactly one nanowire for each signal assignment on the micro-scale wires [DeHon 2005] (Figure 1). This structure can be built by using different axial doping on nanowires. Physical dimensions of these regions can be controlled by adjusting the growth rate. Therefore, a nanowire decoder can be built by using nanowires that have unique addresses encoded in their axial profile.

The second proposal [Ziegler and Stan 2003] uses masking methods to provide one-to-one mapping between microscale wires and nanowires (Figure 2). In this method, a diagonal mask is used to match the micro-scale wire pitch to nanowire pitch.

In Strukov and Likharev [2005], yet another alternative for achieving a CMOS-nanowire interface has been described. In this proposal, a nanowire layer is placed on top of a CMOS plane. The interconnect between these two planes is provided by special pin/pad junctions which are distributed uniformly across the CMOS layer. The nanowire crossbar is slightly rotated so that each nanowire connects to only one pin.

2.2.2 *Interface Fault Models.* The interface methods mentioned above provide various ways to access nanowires using microscale wires. In order to build hybrid CMOS-nano structures, the fault models of the interface have to be considered. Recall that the goal of the interface methods is to establish a one-to-one mapping between the microscale wires and nanowires. However, because of the defects or misalignment of nanowires, there may be one-to-many or one-to-none mappings. In other words, one address assignment on the microscale wires can select multiple nanowires at the same time or cannot activate any nanowires at all. For example, the interface method, which uses a nanowire decoder, addresses each nanowire individually using a coding scheme integrated onto nanowires during fabrication. Although this method is susceptible to the fault models described above, the probability of assigning a unique address to each

nanowire can be increased to over 99% by selecting a coding scheme carefully and using a larger address space than the number of nanowires [DeHon 2005].

3. HYBRID ARCHITECTURE

In this section, we describe our hybrid architecture consisting of both nanowire and CMOS logic elements. The primary goals of the architecture are tolerating high manufacturing defect rates, reducing the general unreliability of computations caused by transient faults, and facilitating fast compilation. Before we introduce the architecture in detail, we identify three basic problems germane to all nanoscale computing systems.

3.1 Problems in Nanoscale Circuits

Circuit design using devices at structure sizes in the range of few nanometers faces several fundamental defect-related problems. We classify them into three categories: *located permanent faults, unlocated permanent faults* and *transient faults*.

—Located permanent faults are permanent manufacturing defects in the chip that have been located during testing. They may be wire defects in the form of broken nanowires or unprogrammable junctions due to stuck-open (stuck-closed) faults in which the junctions are permanently open (closed). DeHon [2005] estimates the probability of broken nanowires and stuck-closed junctions to be negligibly small, and the probability of stuck-open junctions to be around 1–10%.

—Unlocated permanent faults are permanent faults in the chip which are not located during testing. Since testing methods for nanoarchitectures are probabilistic in nature [DeHon 2005; Mishra and Goldstein 2003], it is reasonable to expect that, due to the sheer scale of the chip, a significant fraction of defective junctions and broken nanowires will remain unlocated, causing the chip to malfunction.

—Transient faults are dynamic faults that occur at run-time even if the device is free of permanent manufacturing defects. Transient faults, which lead to soft errors, can be caused by perturbations to the substrate, for example, by subatomic particles causing charge fluctuations, noise or heat [Snider et al. 2005; Rejimon and Bhanja 2006]. In CMOS logic circuits, transient faults have been addressed [Shivakumar et al. 2002; Mukherjee et al. 2005], but do not pose serious problems yet. Nanoscale circuits, on the other hand, are expected to be much more prone to transient faults due to their smaller structures [Snider et al. 2005].

3.2 Architectural Details

The proposed architecture addresses all the above types of faults. The two major ideas involved are: (i) using redundancy in a "hierarchical" manner, and (ii) using majority voting with both nanowire and CMOS components to build a hybrid, fault-tolerant architecture. Other key contributions relate to the compiler and will be discussed in Section 4.

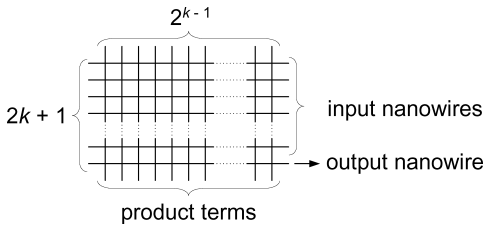


Fig. 3. A k-LUT nanowire processing element.

3.2.1 *Nanowire Processing Element.* The basic unit of our architecture is a nanowire-based crossbar which is formed by intersecting h horizontal and v vertical nanowires. We refer to this structure as a *nanowire processing element (NPE)*. A natural outcome of using NPEs is the easy realization of Boolean look-up tables (LUTs), where a k-input LUT is a logical unit capable of implementing any single-output Boolean function of up to k inputs. We refer to a k-input LUT as a k-LUT for convenience. Although we restrict ourselves to k-LUTs with one output, the framework can be easily extended to allow multi-output LUTs or any other functional representation.

Recall from Section 2 that our nanowire architecture implements diode-based logic. The input signals to the LUTs are provided via h rows. While realizing a function, v columns are used to implement product terms. Therefore, the maximum number of inputs to a LUT is determined by the number of rows as well as the number of columns. For example, to implement a k-LUT, as shown in Figure 3, we need at least $2k + 1$ rows, since one wire for the output and two wires per input (one for the signal and another for its complement) are required. The number of columns should be at least 2^{k-1} because this is the maximum number of product terms required to implement any function with k inputs.

On the other hand, as fabrication of nanowire crossbars is defect-prone, using a minimum number of nanowires for implementing k-LUT NPEs may not be enough to ensure correct implementation of the functions. As a remedy, additional nanowires *within* each NPE can be provided to form the first level of hierarchical redundancy. Having spare rows and columns improves defect tolerance by replacing defective nanowires. On the other hand, redundancy within NPEs is not crucial. Since most k-input functions require less than 2^{k-1} product terms, the remaining nanowires can also be used to enhance defect tolerance.

3.2.2 *Replication and Majority Voting.* The second level of hierarchical redundancy is obtained by replicating each NPE several times and using a majority voter to compute the final result. For redundancy, we may consider different schemes, such as triple modular redundancy (TMR) and five-fold modular redundancy (FMR). At this point, an example will be helpful to understand the effect of the redundancy scheme on the error probability of the circuit. If we define p to be the probability of a transient fault at the output of a single NPE, then at least two NPEs should have transient faults simultaneously to cause an error at the output of TMR. Therefore, the probability of TMR computing an incorrect result is $3p^2 + p^3$, that is $O(p^2)$. It is straightforward to extend

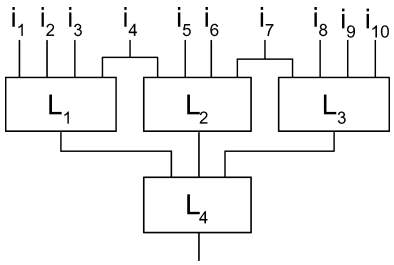


Fig. 4. Dependence of LUT inputs.

this result for a scheme which replicates an NPE $2d + 1$ times. This scheme reduces the error rate from p to $O(p^{d+1})$.

Although the above discussion concludes that redundancy in terms of replicating NPEs will help reduce the probability of incorrect computation, the inherent assumption of error-free majority voters limits its applicability to nanowires. Furthermore, the effect of using redundancy schemes in a circuit, rather than a single logic element, is more complicated and the exact calculation of the probability of observing an incorrect output for a given transient fault rate is NP-hard [Rejimon and Bhanja 2006]. However, to analyze the effect of using nanowire voters and varying number of replications on the probability of observing a fault at the primary outputs of a circuit, we used theoretical and empirical approaches.

Theoretical Approach. Since exact theoretical output error probability calculation is NP-hard, we assume that the inputs to the LUTs are independent. For example, Figure 4 shows a very simple circuit with ten primary inputs, four LUTs, and one primary output. In our theoretical transient error probability calculation, we assume that primary inputs are independent of each other. Furthermore, it is assumed that the inputs of LUT L_4 are also independent. Since LUTs $L_1 - L_2$ and $L_2 - L_3$ share common inputs, this assumption does not hold. However, it makes computation of the probability tractable for very large circuits. It must be noted that the values obtained by the theoretical approach are compared with the ones generated by the empirical approach later on.

With the assumption of the independence of LUT inputs, there are several approaches for calculating the probability of observing an error due to a transient fault at the output of a LUT. One possible approach is to create a tree diagram. Figure 5 shows an example tree diagram for a k-LUT. While creating the tree diagram, first, one of the inputs is selected (i_1). The input can have two values, namely, 0 or 1. Then, there is a probability that these values are correct (C) or faulty (F). For instance, for the first input, if we analyze the 0 branch, the C branch represents the case where the observed and actual values are 0, whereas the F branch demonstrates the case of a transient fault and although 0 is observed, the actual value is 1. By expanding the tree for all inputs, it is possible to obtain all possible actual and observed input value combinations at the leaf nodes. The probability of reaching a leaf node is simply the product of the probability values along the path from the root to the leaf node. Finally, the

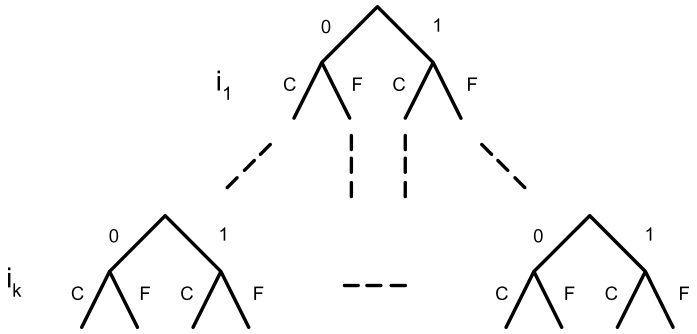


Fig. 5. Tree diagram used to calculate the probability of observing a transient fault at the output of a k-LUT.

leaf nodes can be classified as correct or incorrect by comparing the value of actual and observed output values. Consequently, the probability of observing an incorrect output because of the faults at the LUT inputs will be:

$$P(incorrect\_LUT\_output) = \sum P(faulty\_leaf\_nodes). \tag{1}$$

However, (1) does not include the effect of transient faults at the LUT output. For example, if a transient fault causes a bit flip at the LUT output, it can either change a correct output value to an incorrect one, or correct an incorrect output value caused by the faults at the inputs. Therefore, the probability of observing an incorrect output because of the faults at LUT inputs is not equal to the probability of observing a transient fault at the LUT output. To formalize this discussion, if we define p to be the probability of a transient fault at the output of a LUT, then:

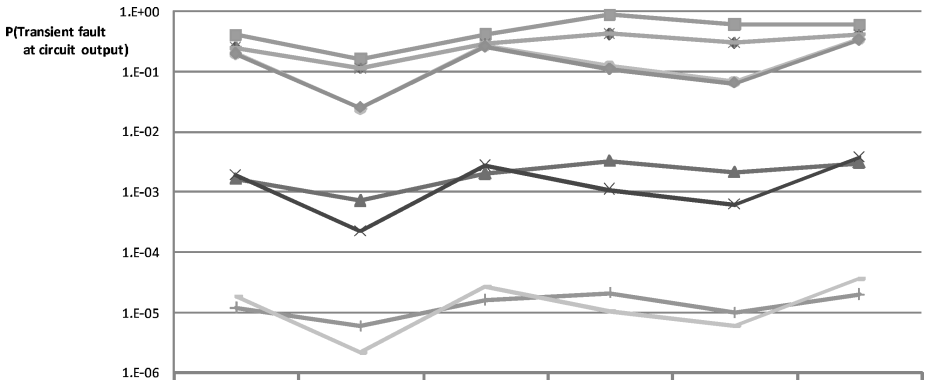
$$\begin{aligned} P(faulty\_LUT\_output) = {} & P(incorrect\_LUT\_output) * (1-p) \\ & + (1 - P(incorrect\_LUT\_output)) * p. \end{aligned} \tag{2}$$

When replication is used, if the voter is fault-free, a faulty output is observed only if the majority of the replications produce faulty outputs. As explained before, $2d+1$ replications reduce probability p to $O(p^{d+1})$. On the other hand, if the voter is also susceptible to transient faults, then it can be regarded as a LUT, and the equations presented above can be used to find its probability values.

Finally, after the fault probabilities for each LUT in a circuit is calculated, the probability of observing a transient fault at output of each primary output's voter can be calculated using the previous formulas. If we denote this probability by $P(faulty\_primary\_output)$, then the probability of observing transient faults at the circuit output can be estimated by:

$$P(faulty\_circuit\_output) = 1 - \prod (1 - P(faulty\_primary\_output)). \tag{3}$$

Empirical Approach. In order to test the accuracy of the theoretical calculations, a set of simulations was performed. Circuits with no redundancy and circuits using TMR and FMR with CMOS and nanowire majority voting logic were simulated. The probability of a transient fault at the output of an NPE



	186	262	314	395	461	529
No F.T. (sim)	2.46E-01	1.12E-01	2.84E-01	4.23E-01	2.96E-01	4.03E-01
NO F.T. (theory)	4.08E-01	1.62E-01	4.10E-01	8.73E-01	6.07E-01	5.91E-01
TMR CMOS (sim)	1.68E-03	7.29E-04	2.03E-03	3.32E-03	2.16E-03	3.10E-03
TMR CMOS (theory)	1.86E-03	2.21E-04	2.69E-03	1.09E-03	6.05E-04	3.67E-03
TMR NANO (sim)	2.48E-01	1.13E-01	2.86E-01	4.25E-01	2.98E-01	4.05E-01
TMR NANO (theory)	1.95E-01	2.42E-02	2.63E-01	1.20E-01	6.68E-02	3.41E-01
FMR CMOS (sim)	1.20E-05	6.00E-06	1.60E-05	2.10E-05	1.00E-05	2.00E-05
FMR CMOS (theory)	1.80E-05	2.15E-06	2.66E-05	1.05E-05	5.92E-06	3.63E-05
FMR NANO (sim)	2.46E-01	1.12E-01	2.84E-01	4.24E-01	2.97E-01	4.03E-01
FMR NANO (theory)	1.95E-01	2.47E-02	2.57E-01	1.10E-01	6.40E-02	3.34E-01

Fig. 6. Comparison of the effect of fault tolerance methods evaluated by simulations and theoretical approach.

was assumed to be 10^{-2} per cycle per bit. It must be noted that the probability of transient faults was set to a very high value to be able to observe transient faults within the simulation time. The circuits were simulated for only one million cycles because of CPU time constraints. Since a nanowire-based majority voter is a special kind of an NPE, it has the same probability of transient faults as the other NPEs. On the other hand, as transient faults do not pose serious problems for CMOS logic elements, it is assumed that CMOS majority voters are error-free. Under these assumptions, the simulations were carried out for a range of designs having 186 up to 529 NPEs.

The results of the simulations and theoretical calculations are shown in Figure 6. In the figure, the "No F.T." lines present results for the no fault tolerance case, that is, for original circuits with no replications. The names for other lines start with a three-letter abbreviation of the redundancy scheme followed by the type of voter, for example, nanowire-based or CMOS. Finally, the method used to generate the value, namely, simulations or theoretical approach, is given in parenthesis.

An initial analysis can be made of the accuracy of the theoretical approach. For most cases, the values generated by theoretical analysis are comparable to those of the empirical approach. Therefore, for larger circuits and more realistic, hence much smaller, probability of transient fault values for NPEs, the theoretical approach can be used.

Next, the effect of using nanowire-based and CMOS voters can be compared. The figure demonstrates that using TMR and FMR with nanowire-based

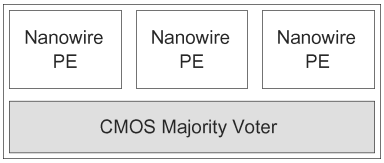


Fig. 7. Hybrid nano-CMOS computing element example.

majority voters produces results similar to the cases with no fault tolerance due to the fact that the nanowire majority voting logic itself could be error-prone. The lines representing these cases overlap in the figure. This implies that regardless of the replication strategy, for example, TMR or FMR, using nanowire-based voters does not increase fault tolerance of the original design. Therefore, using CMOS for voting is essential to correct transient faults.

Finally, the effect of using TMR and FMR with CMOS voters can be examined. The figure illustrates that the results are as expected. TMR reduces the fault probability p to p^2 and, similarly, FMR p to p^3.

Although using CMOS voters is motivated by the simulations and theoretical calculations, it has a high area overhead. However, keeping the CMOS logic elements simple will decrease the area of the overall structure, as we will see next.

3.2.3 *Building Blocks and Overall Architecture.* The basic building block of the proposed architecture is composed of a set of replicated NPEs followed by a majority voter constructed using CMOS. Building this structure is possible by using one of the CMOS-nanowire interface methods mentioned in Section 2.2. The CMOS voter corrects errors due to transient faults and unlocated permanent faults.

Figure 7 shows three NPEs followed by a CMOS majority voter. A generalized version of Figure 7 is the building block of the hybrid architecture. The architecture consists of a two-dimensional array of *hybrid clusters*. Each hybrid cluster is characterized by three parameters: n, r, and m. An (n, r, m) hybrid cluster consists of n fully interconnected NPEs, each replicated r times, and m CMOS voters. m and n allow us to trade-off fault tolerance for area: if m is large (m cannot be greater than n since we need at most one CMOS voter for one redundant set of NPEs), we achieve good fault tolerance at the expense of a comparatively large CMOS area. The implementation details of using fewer voters than total number of NPEs, and its trade-off analysis is presented in Section 3.2.5.

Figure 8(a) shows an $(n, 3, m)$ hybrid cluster. Using NPEs within clusters makes it possible to use redundancy schemes that are considered very area-expensive fault-tolerant techniques in the CMOS world. Since nanoscale manufacturing processes lead to highly regular structures, we assume that all NPEs within a hybrid cluster are identical. A given function is mapped onto each copy separately (if such a mapping is possible) by routing around the defects, and the output is fed to a CMOS majority voter.

The overall nano-CMOS architecture is a 2D array of hybrid clusters with N_V rows and N_H columns (see Figure 8(b)). Each hybrid cluster accepts inputs from other clusters, and provides outputs to other clusters. Since errors

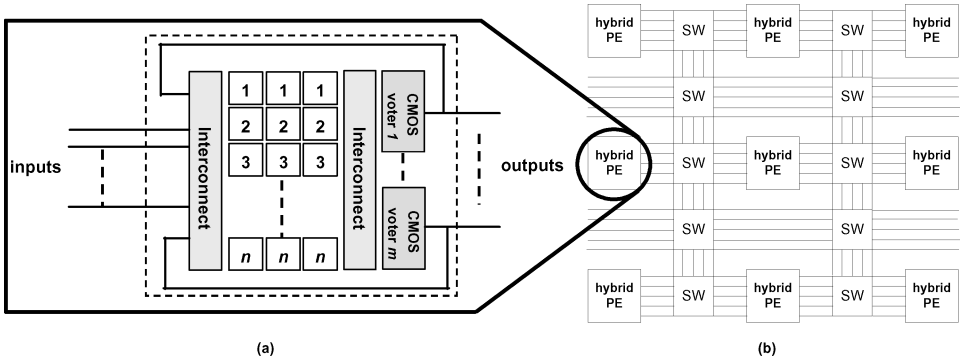


Fig. 8. (a) An $(n, 3, m)$ hybrid cluster, and (b) the overall hybrid architecture.

caused by faults in nanowire interconnects between clusters are difficult to deal with, we assume that the intercluster interconnect is realized using metal. For this reason, each switching block (SW), which provides signal routing on metal wires, is implemented using programmable bidirectional tri-state buffers [Schmit and Chandra 2005]. However, as mentioned above, nanowires are used to implement logic. By using NPEs, we can apply redundancy schemes without significant area overhead; whereas, if CMOS were used for implementing processing elements, it would be very costly in terms of area.

3.2.4 *Area Overhead of Using CMOS Voters within Hybrid Clusters.* At this point, it will be instructive to compare the area of a hybrid cluster which uses CMOS voters to a hybrid cluster which uses nanowire voters. Although the discussion presented in Section 3.2.2 concludes that using CMOS voters is crucial for soft error tolerance, this comparison can be used to identify the trade-off between fault tolerance and area.

In order to calculate the area of a hybrid cluster approximately, counting the number of horizontal and vertical nanowires is necessary. Since hybrid clusters have a very regular structure, this is possible by examining each component of the architecture one by one, namely the NPEs and the interconnection network. The number of nanowires required for implementing a k-LUT NPE was presented in Section 3.2.1. Similarly, since the intra-cluster routing channels and the interconnection crossbars are also realized in nanowires, a similar approach can be followed to count the number of nanowires necessary for routing.

Figure 9 shows an implementation of a $(4, 3, 2)$ hybrid cluster that uses k-LUT NPEs. The number of nanowires for the routing channel underneath each redundant set of NPEs is $2k + 3$, where $2k$ nanowires are used to route the input signals and the remaining three nanowires are required for routing the outputs of the replications of the NPEs. It is possible to follow the same approach to find the number of nanowires for the crossbar that routes the input signals. Since there are four sets of NPEs, each of which requires $2k$ input signals, a total of $(2k) * 4$ input nanowires are required. The crossbar located just before the voters routes the output signals as well. Therefore, a total of $(2k + 3) * 4$ nanowires are necessary.

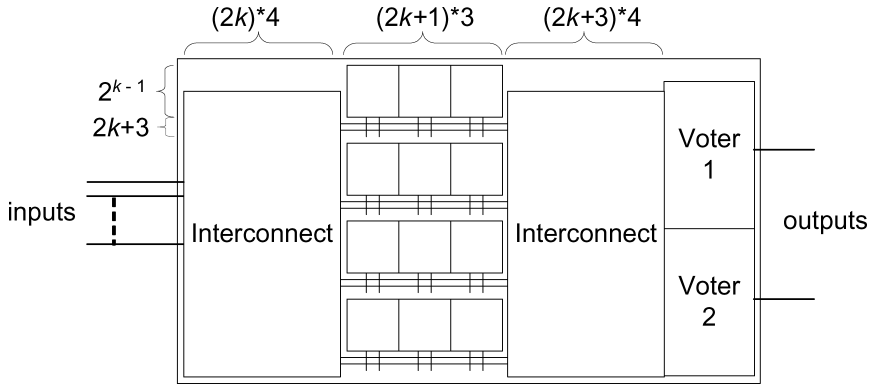


Fig. 9. Minimum number of nanowires required for a (4, 3, 2) hybrid cluster that contains k-LUT NPEs.

Consequently, a hybrid cluster requires at least $((2^{k-1} + (2k + 3)) * 4)$ horizontal nanowires and $(2k * 4 + (2k + 1) * 3 + (2k + 3) * 4)$ vertical nanowires. It is important to note that this calculation does not include the dimensions of the voters and this is the only difference between a hybrid cluster that uses CMOS voters and the one that uses nanowire voters.

We calculated the dimensions of a CMOS voter to be 2.2μm$\times 2.4\mu$m using the 65nm technology library provided by Taiwan Semiconductor Manufacturing Company[1]. Since the technology node for CMOS does not reflect the theoretical minimum size, we decided to use $15nm$ as the nanowire diameter and $30nm$ as the nanowire pitch [Snider and Williams 2007], which preserves the ratio of the used CMOS technology node dimension to the theoretical limit. On the other hand, the implementation of a nanowire voter requires a 4×3 crossbar.

When we plug in the values for 3-, 4-, 5-, and 6-LUT implementations, the area overhead of using CMOS voters can be calculated as 219%, 102%, 48% and 41%, respectively. The huge overhead for 3- and 4-LUT implementations stems from the fact that the dimensions of the nanowire components in a hybrid cluster are dwarfed by the dimensions of the two CMOS voters. This result implies that the overhead of the CMOS voters can be reduced by using more complex NPEs.

Another comparison can be made between the area of our hybrid cluster and the area of a pure CMOS implementation. For example, for the 4-LUT implementation, the above comparison implies that a (4, 3, 2) hybrid cluster has the same area as four CMOS majority voters. This means that our architecture is capable of computing four different 4-LUT functions with TMR within the area of just four CMOS majority voters. This example shows the area advantage of our architecture over a pure CMOS implementation.

Finally, the effect of varying the number of CMOS voters in a cluster on the area can be analyzed. The (4, 3, 2) hybrid cluster, which was analyzed above, has only two voters. For example, if we compare the area of a cluster with four CMOS voters, that is, a (4, 3, 4) hybrid cluster to the (4, 3, 2) hybrid cluster, then

[1]www.tmsc.com.

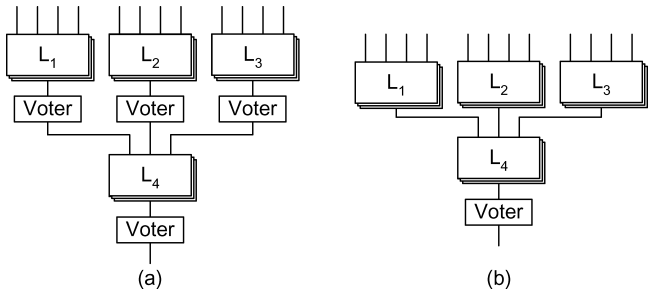


Fig. 10. Voting schemes for the NPEs in a cluster: (a) at the output of each NPE, and (b) at the output of the DAG.

the area overhead for 3-, 4-, 5-, and 6-LUT implementations can be calculated as 30%, 28%, 26%, and 24%, respectively. The results demonstrate that, on top of keeping CMOS logic simple, reducing the number of CMOS elements helps reduce the area significantly. Next, we present a technique to reduce the number of CMOS voters in a cluster.

3.2.5 *Computation Using Directed Acyclic Graphs.* It is possible to reduce the number, hence the area, of CMOS elements by organizing computations in the form of a directed acyclic graph (DAG) of NPEs with voting only performed at the output of the DAG. The complete DAG can then be replicated, and outputs of these replications can be checked for faults using CMOS voters. Figure 10 shows the difference between the ideal computation and the one with DAG. The ideal computation, shown in Figure 10(a), has a voter at the output of each redundant set of NPEs; therefore, in a hybrid cluster, the number of voters is equal to the number of redundant set of NPEs. On the other hand, if the implementation of the same function is realized in the form of a DAG, as shown in Figure 10(b), only the output of the replicated DAG needs to be voted on. For the example given in the figure, the DAG method requires only one voter compared to four voters in the original method. As the located defects are avoided during mapping of functions onto NPEs, application of the DAG method does not affect defect tolerance. However, as explained in the previous section, it offers a significant area advantage.

Let us now consider the effect of the DAG method on fault tolerance. Since not all NPEs are voted on, the circuit outputs of the DAG implementation is expected to be affected by the transient faults more than that of the original implementation. We used the theoretical analysis presented in Section 3.2.2 to compare these two cases by using $(4, 3, 2)$ and $(4, 3, 4)$ hybrid clusters, in which there are two and four CMOS voters, respectively. In the calculations, the probability of a transient fault at the output of an NPE is assumed to be 10^{-2}. Figure 11 shows the calculated fault probabilities for no fault tolerance, and TMR and FMR using CMOS voters. The measurements shown in the figure suggests that, although the DAG method decreases the probability of observing transient faults at the primary outputs, it does not reach the levels achieved by using the original method. Furthermore, if the DAG method is used, using FMR instead of TMR does not increase fault tolerance.

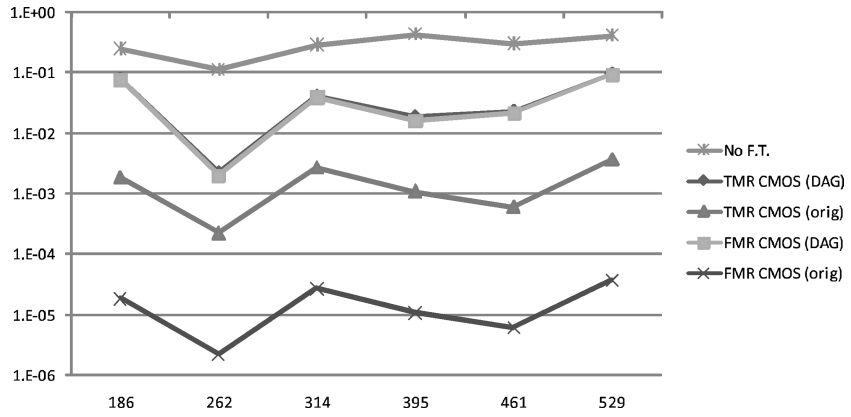


Fig. 11. The effect of using DAG on fault tolerance.

Although application of the DAG method decreases the area of a hybrid cluster, it increases the probability of observing transient errors at the primary outputs. However, if area is the main concern in a design, it is still possible to increase fault tolerance by sharing the CMOS voters over multiple cycles. For example, if a hybrid cluster implements four LUTs and has only two voters, the computation can be divided into two cycles. In the first cycle, two LUTs will use the voters, and in the second, the remaining LUTs. This method makes it possible to apply the DAG method and have no adverse effect on fault tolerance. However, the path delays in the circuit will increase. Consequently, the DAG method allows trade-offs among area, fault tolerance and path delays.

4. HYBRID ARCHITECTURE COMPILER

The function of the hybrid architecture compiler is to map circuits to the nano-CMOS hybrid architecture. In this section, we present details of the compilation flow, including the algorithms used. The compiler has a technology-independent portion that is unaware of manufacturing defects, and a technology-dependent portion that uses an optional defect map. The defect map specifies defects that were located during testing. Note that the defect map does not need to be complete (defects not specified in the defect map will be treated as unlocated permanent defects).

As stated in Section 1, it is difficult to obtain an exact defect map for manufactured chips. This problem was addressed in Tahoori [2005]. By looking at the expected defect rates of the manufacturing process, a method was suggested to identify defect-free subsets within the original partially-defective crossbars. However, the assumption that functions should be implemented on defect-free subsets is an overkill since it is possible to map functions onto partially defective crossbars.

Figure 12 shows the major parts of the compiler. The three main inputs to the compiler are: (i) a circuit description in the Berkeley Logic Interchange Format (BLIF), (ii) a set of architectural parameters (the important ones are shown in the figure), and (iii) an optional defect map. The flow starts with logic

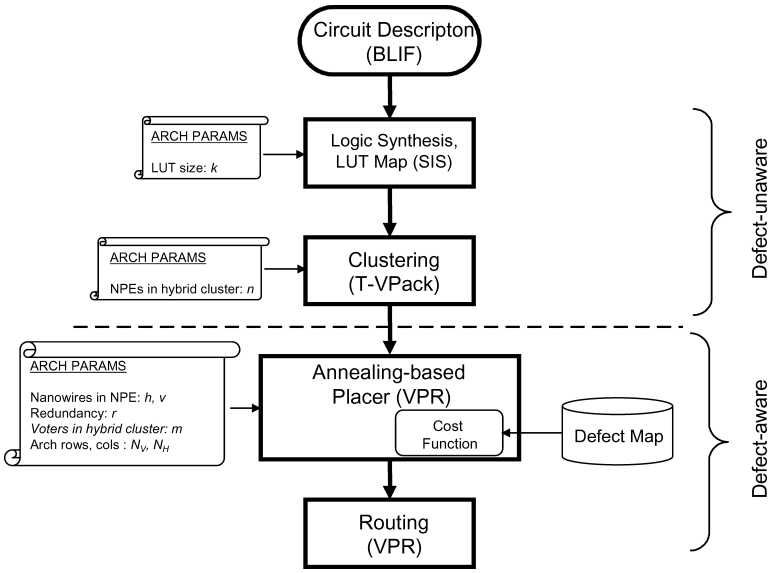


Fig. 12. Hybrid architecture compiler.

synthesis, which is used to map circuit description onto LUTs. Next, depending on the parameters of the architecture, the LUTs are packed together to form hybrid clusters. These steps constitute the defect-unaware part of the flow. The following steps are defect-aware since the defect map information is used by the placer. When the placement information is ready, the flow is completed by routing the design. We explain each part of the compiler in detail next.

4.1 Logic Synthesis and Clustering

The logic netlist is first synthesized and mapped to k-LUTs. We use SIS [Sentovich et al. 1992] for logic synthesis, and the built-in PLD package for LUT mapping. The synthesized and LUT-mapped circuit, which is a network of k-LUTs, is then grouped into hybrid clusters using T-VPack [Betz and Rose 1997]. The clustering process tries to pack as many k-LUTs as possible into a single (n, r, m) hybrid cluster. In particular, T-VPack tries to maximize intracluster communication, thereby minimizing intercluster routing. This is crucial because significant delay and power dissipation are incurred in intercluster routing.

In the compiler, both logic synthesis and clustering are defect-unaware. We defer the use of defect knowledge to the subsequent placement phase, as described next. In this sense, our compiler is easily retargetable to different defect models.

4.2 Defect-Aware Placement Strategy

The output of the clustering phase is a network of k-LUT clusters where each cluster has no more than n LUTs (recall n is the number of NPEs in an (n, r, m) hybrid cluster). Other inputs to the placer, which is based on VPR [Betz and

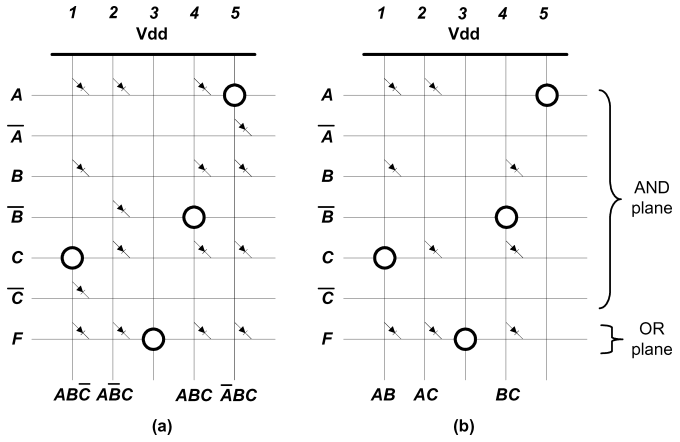


Fig. 13. Two realizations of a three-input majority function.

Rose 1997], are relevant architectural parameters, such as the number of CMOS voters in a hybrid cluster (see Figure 12 for a complete list), and the optional defect map. The output of the placer is a netlist in which each LUT cluster in the output of the clustering phase is mapped to a physical hybrid cluster.

The defect map contains information about broken nanowires and defective junctions. For a particular chip, this information can be collected using one or more of the testing methodologies proposed for nanowire-based circuits [Mishra and Goldstein 2003; DeHon and Naeimi 2005; Brown and Blanton 2004; Jacome et al. 2004]. If a defect map is not provided, placement onto a defect-free physical chip is assumed.

Checking the feasibility of mapping a cluster of LUTs to a physical, possibly defective, hybrid cluster can be broken down into two parts: (i) mapping a k-LUT to a defective NPE, and (ii) mapping a cluster of k-LUTs to a defective (n, r, m) hybrid cluster. We describe each of these steps next.

4.2.1 *Mapping a k-LUT to an NPE.* Using diode junctions, it is possible to construct AND-OR planes, as shown in Figure 13. Input signals are fed to horizontal nanowires, while the vertical nanowires are configured to implement product terms by activating appropriate junctions. Junctions on the output horizontal wire are activated to implement the OR-plane.

Since we provide all input signals in true and complemented form, it is possible to implement any k-input Boolean function using an NPE. Figure 13(a) shows an unoptimized implementation of a three-input majority gate that uses every minterm, and Figure 13(b) shows the same function after minimizing the product terms. This example illustrates the use of "functional redundancy," which we describe in the next subsection.

In practice, it may not be possible to map any k-input Boolean function to an NPE due to defects. For instance, a broken nanowire cannot be used at all, neither can nanowires involved in junctions with stuck-closed defects. On the other hand, a nanowire with the more common stuck-open junction defect can be used in conjunction with other nanowires that do not a have stuck-open

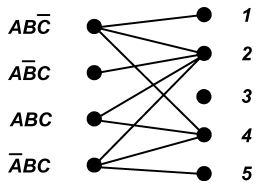


Fig. 14. Bipartite graph for Figure 13(a).

defect on corresponding junctions. For example, in Figure 13, the function is mapped in such a way that it avoids all stuck-open defects (shown by circles), and uses an extra vertical nanowire.

We use bipartite matching to decide if a k-LUT can be mapped to a defective NPE. A bipartite graph is an undirected graph whose vertices can be partitioned into two sets such that no edge connects vertices in the same set (Figure 14). While creating the bipartite graph, we add a node on the "left side" for each product term, and a node on the "right side" for each vertical wire (these wires are numbered as shown in Figure 13).

We now pick an assignment of input signals to horizontal wires (pin assignment). We do not attempt to find an optimal pin assignment since that is provably NP-hard. Instead, we assign pins based on a heuristic that we will soon explain. Given a pin assignment, an edge between a product term t and a vertical nanowire w indicates that t can be realized by w. Figure 14 shows the bipartite graph corresponding to Figure 13(a). The existence of a matching in the bipartite graph of a function implies that the function can be realized. On the other hand, if there are unmatched product term nodes, then bipartite graph construction and matching is repeated with another pin assignment. After a reasonable number of pin assignments, we deem the function as not mappable, and proceed to attempt mapping on a different hybrid cluster.

During placement, functions will be mapped to defective NPEs several times. Hence, the above process must be fast. Our fast heuristic, presented in Figure 15, begins with pin assignment (steps 1–5). To choose the pin assignment, we assign the most frequently used signals in the product terms to the nanowires with the smallest number of defects. Next, instead of constructing the complete bipartite graph, which is time-consuming, we start by finding a single edge for each product term. This part is shown in steps 8–16. While searching for mappings for the product terms, the vertical nanowire nodes that have stuck-open defects on the junction of the output nanowire are skipped. The remaining ones are checked to see if they can realize the product term, starting with the ones with the least number of edges. If the first vertical nanowire node that can realize the product term has other edges, it implies that there are other product terms using this nanowire. Those product term nodes are added to the search list. On the other hand, if it is not possible to find a vertical nanowire that can implement a product term, the algorithm terminates by concluding that the mapping is not possible.

Once an edge for each product term is added to the bipartite graph, a matching algorithm is run to check if given assignments form a valid mapping. The heuristic terminates once a valid mapping is found. On the other hand, if a

```
 1: for all literals in the function do
 2:     count number of appearances
 3: for all horizontal nanowires do
 4:     count number of defects on the junctions
 5: Pin assignment: nanowires with fewest defects for most frequently used literals
 6: add nodes to bipartite graph
 7: add all product term nodes to search list
 8: for each product term node do
 9:     remove the node from search list
10:     sort vertical nanowire nodes with increasing number of edges
11:     check mapping until one mapping is found
12:     if no mapping is available then
13:         return no matching is possible
14:     insert an edge between the product term node and nanowire node
15:     if nanowire node has other edges then
16:         add all neighboring product term nodes to search list
17: repeat
18:     check for a matching
19:     if matching exists then
20:         return matching information
21:     pop a product term node from search list
22:     check mapping onto vertical nanowires until one mapping is found
23:     if no new mappings are available then
24:         goto until
25:     if nanowire node has other edges then
26:         add all neighboring product term nodes to search list
27: until all possible mappings are checked or the search list is empty
28: restart the search with another pin assignment
```

Fig. 15. A heuristic for mapping a k-LUT to a defective NPE.

match is not found, the search for new edges continues by choosing product term nodes from the search list (steps 17–27). These steps are repeated until a valid match is found or all the nodes in the search list are checked. At the end, if no match is possible, we restart the search with a different pin assignment.

Although logic mapping heuristics for nanowire crossbars exists [Rao et al. 2006], they are not applicable to our scenario. Since we use the mapping heuristic within the inner loop of placement, it has to be very fast. For this reason, instead of creating a complete bipartite graph, we construct it step by step. Similarly, we propose our own heuristics to map LUTs to a hybrid cluster, as explained next.

4.2.2 *Mapping LUTs to a Hybrid Cluster.* Our goal here is to map a network of k-LUTs (the output of the clustering phase) to a physical hybrid cluster. To do this, we use the heuristic presented in Figure 15 to determine if each k-LUT can be mapped to a candidate NPE within the cluster. We construct another bipartite graph with a node for each k-LUT on the left side, and a node for each set of redundant NPEs on the right side. The fast heuristic for checking a valid cluster mapping is similar to the heuristic used in Section 4.2.1. When a match is found, a valid cluster mapping exists. On the other hand, if any function node is left unmatched after all trials, the algorithm terminates unsuccessfully.

To improve reliability, each function is mapped to redundant NPEs by choosing different realizations. For example, one implementation can use minterms without any simplifications and another can implement the same function using

```
 1: initialize: num invalid assignments = 0
 2: for all clusters in circuit definition do
 3:     choose an empty location randomly
 4:     run mapping heuristic
 5:     if not a valid assignment then
 6:         increment num invalid assignments
 7: start simulated annealing search
 8: for all swaps in the search do
 9:     if move cluster from invalid to valid mapping then
10:         decrement num invalid assignments
11:     else if move cluster from valid to invalid mapping then
12:         increment num invalid assignments
13:     calculate cost function
```

Fig. 16. Modified algorithm for placement.

the minimum sum-of-products form. If we use the same realization for all copies, then for a given input vector, if one copy produces an incorrect output, the probability of other copies having incorrect outputs will be higher; this is known as common-mode or common-cause failure [Mitra et al. 2000]. By choosing different realizations, we aim to reduce the effect of common-mode failures. This is the concept of functional redundancy that we alluded to earlier.

4.2.3 *Changes in the Placement Algorithm.* The simulated annealing-based placement method proceeds as follows (see Figure 16). First, all clusters are randomly "assigned" to specific hybrid clusters on the chip. The heuristic for checking the validity of mapping is run to count the number of invalid mappings. An assignment is declared invalid if the LUT cluster cannot be mapped to a physical cluster due to defects. Then at each temperature, random swaps are performed and the number of invalid mappings is updated incrementally. This is done to avoid running the mapping heuristic for the whole chip. Then the cost of the mapping is calculated. The cost function itself is a key contribution using which we make the placer defect-aware. The cost function C of a placement P is given by:

$$C(P) = C_{VPR} + num\_invalid\_mappings * N_{large}, \tag{4}$$

where C_{VPR} is the timing and area cost function used by VPR [Betz and Rose 1997], and $num\_invalid\_mappings$ the number of infeasible assignments to a physical hybrid cluster. We choose N_{large} to be a large positive number to drive the annealer towards a feasible solution.

Finally, since we use metal for intercluster routing, we use an unmodified third-party router (VPR) together with our placer.

5. EXPERIMENTAL RESULTS

In this section, we evaluate the hybrid architecture using our compiler. We vary the architectural parameters in the flow of Figure 12 to elicit useful information about the performance of the architecture under different defect rates. In particular, we show the effect of defect rates on critical path delay, chip size, and compilation time by using (4, 3, 2) hybrid clusters (recall from Section 3.2 that these hybrid clusters have two CMOS voters and four NPEs with triple

Table I. Statistics of Benchmarks

Circuit	3-LUT		4-LUT		5-LUT		6-LUT	
	#LUTs	#clusters	#LUTs	#clusters	#LUTs	#clusters	#LUTs	#clusters
c2670	350	88	272	68	235	59	220	55
c3540	575	144	389	98	306	77	265	67
c5315	650	163	517	130	397	100	342	86
c7552	653	164	516	129	413	104	384	96
dalu	602	151	379	95	323	81	257	65
des	1631	408	1155	289	962	241	781	196
i8	533	134	364	91	302	76	250	63
i10	976	244	721	181	610	153	529	133
k2	477	120	372	93	321	81	284	71
too_large	212	53	152	38	131	33	108	27

modular redundancy). For the experiments, using TMR is assumed to be enough to prevent circuit failures caused by transient faults. However, it is straightforward to incorporate more redundancy into hybrid clusters, for example, FMR, if effects of transient faults are shown to be greater at the nanoscale. Moreover, for the hybrid clusters, a minimum number of rows and columns is selected in order to analyze the effect of redundancy within an NPE.

Since manufacturing processes for nanowire chips are not yet mature, for the experiments, we create a random defect map based on defect probabilities that have been observed so far [DeHon 2005]. We vary the probability of stuck-open defects (P_{so}) between 1% and 10% and assume probabilities of broken nanowire (P_{bnw}) and stuck-closed (P_{sc}) defects to be zero. Furthermore, to statistically decide the minimum chip size required to implement a circuit, we generate several random defect maps for each defect rate. If 5% of the compilations for a specific defect rate fail, we increase the number of rows and columns of hybrid clusters on the chip and repeat the procedure.

We compiled several MCNC and ISCAS'85 benchmarks to the hybrid architecture by using 3-, 4-, 5-, and 6-LUT NPEs. Table I shows the statistics of the benchmarks. In this table, the second column shows the number of LUTs required to realize each circuit while the third column denotes the number of hybrid clusters used for a 3-LUT implementation. The same statistics are presented for the 4-, 5-, and 6-LUT implementations in the following columns. The statistics show that, in general, as the number of inputs to LUTs increases, fewer LUTs, hence fewer clusters, are required, as expected.

Simulations were performed on the benchmarks to obtain critical path delays, chip sizes and compilation times. Tables II, III, IV, and V show experimental results for $(4, 3, 2)$ hybrid clusters using 3-, 4-, 5-, and 6-LUT NPEs, respectively. In these tables, the array size of defect-free implementation, in terms of the number of rows and columns of hybrid clusters, is given in the second column. The following columns denote critical path delay, array size and compilation time for 1%, 5%, and 10% P_{so} values. The delay values are calculated by using the nanowire models given in [DeHon 2005], and 32nm CMOS models provided by the predictive technology model [PTM 2007]. All compilation times are on a 2.2GHz PC with 1GB DRAM running under Linux kernel version 2.4.21-20.

Table II. Experimental Results for 3-LUT Networks

Circuit		$P_{so} = 1\%$			$P_{so} = 5\%$			$P_{so} = 10\%$		
	array size (N_H, N_V)	delay (ns)	array size (N_H, N_V)	compile time (s)	delay (ns)	array size (N_H, N_V)	compile time (s)	delay (ns)	array size (N_H, N_V)	compile time (s)
c2670	10, 9	2.907	10, 9	28.8	2.907	10, 9	29.1	2.907	10, 9	29.0
c3540	12, 12	3.900	12, 12	46.9	3.903	12, 12	46.9	3.903	12, 12	46.8
c5315	13, 13	2.570	13, 13	67.6	2.570	13, 13	66.9	2.570	13, 13	67.4
c7552	13, 13	3.390	13, 13	75.1	3.390	13, 13	74.3	3.390	13, 13	74.3
dalu	13, 12	5.084	13, 12	48.1	5.080	13, 12	48.2	5.080	13, 12	47.9
des	21, 20	2.560	21, 20	201.9	2.560	21, 20	201.8	2.560	21, 20	203.8
i8	12, 12	1.710	12, 12	53.5	1.710	12, 12	53.6	1.710	12, 12	53.7
i10	16, 16	4.480	16, 16	118.4	4.480	16, 16	118.6	4.480	16, 16	118.2
k2	11, 11	1.871	11,11	36.2	1.871	11, 11	36.3	1.871	11, 11	36.1
too_large	8, 7	1.837	8, 7	27.7	1.840	8, 7	27.8	1.840	8, 7	27.7

Table III. Experimental Results for 4-LUT Networks

Circuit		$P_{so} = 1\%$			$P_{so} = 5\%$			$P_{so} = 10\%$		
	array size (N_H, N_V)	delay (ns)	array size (N_H, N_V)	compile time (s)	delay (ns)	array size (N_H, N_V)	compile time (s)	delay (ns)	array size (N_H, N_V)	compile time (s)
c2670	9, 8	2.166	9, 8	34.7	2.170	9, 8	44.8	n/a	n/a	n/a
c3540	10, 10	2.609	10, 10	44.4	2.617	10,10	49.7	n/a	n/a	n/a
c5315	12, 11	2.201	12, 11	74.4	2.196	12, 11	96.2	n/a	n/a	n/a
c7552	12, 11	2.997	12, 11	85.2	3.006	12, 11	143.9	3.022	14, 14	136.2
dalu	10, 10	4.290	10, 10	40.8	4.290	10, 10	40.9	4.292	10, 10	40.5
des	17, 17	2.250	17, 17	206.2	2.250	17, 17	207.1	2.242	17, 17	208.5
i8	10, 10	1.510	10, 10	49.8	1.510	10, 10	49.4	1.510	10, 10	48.8
i10	14, 13	3.735	14, 13	130.2	3.763	14, 13	145.7	3.751	16, 15	124.3
k2	10, 10	1.650	10, 10	36.0	1.650	10, 10	36.2	1.645	10, 10	36.0
too_large	7, 6	1.520	7, 6	28.7	1.520	7, 6	28.7	1.518	7, 6	28.7

The experiments indicate that the critical path delay typically increases slightly with an increasing defect rate. This result is expected as it may not be possible to place clusters optimally due to invalid mappings caused by defects. However, it is useful to note that the defect rate does not drastically affect critical path delay.

The average compilation time is not affected by the defect rate if it is possible to find mappings to the same array size for different defect rates. On the other hand, if extra hybrid clusters are used for redundancy, then compilation time typically increases slightly. This is because the placer has more candidates to consider during each iteration of simulated annealing. However, since the heuristic is designed to be fast, the compilation times are also not affected much by increasing defect rates.

The array size, on the other hand, does not increase with increasing defect rates for 3-LUT NPEs. However, for 4-, 5-, and 6-LUT NPEs, some of the circuits require more NPEs than a defect-free realization would require. This implies that redundancy in terms of extra hybrid clusters was enough to successfully map to defective chips. Nevertheless, it is not possible to realize some circuits under 5% and 10% defect rates at all. This implies that NPEs are fully utilized and no extra nanowire is left for redundancy. This result suggests that

Table IV. Experimental Results for 5-LUT Networks

Circuit	array size (N_H, N_V)	$P_{so} = 1\%$ delay (ns)	array size (N_H, N_V)	compile time (s)	$P_{so} = 5\%$ delay (ns)	array size (N_H, N_V)	compile time (s)	$P_{so} = 10\%$ delay (ns)	array size (N_H, N_V)	compile time (s)
c2670	8, 8	1.699	8, 8	55.4	1.714	12, 13	63.9	n/a	n/a	n/a
c3540	9, 9	2.482	9, 9	47.7	2.493	9, 9	59.1	n/a	n/a	n/a
c5315	10, 10	1.745	10, 10	95.8	1.742	10, 11	144.6	n/a	n/a	n/a
c7552	11, 10	2.372	11, 10	123.0	2.384	13, 12	250.3	n/a	n/a	n/a
dalu	9, 9	2.790	9, 9	48.1	2.790	9, 9	48.1	2.790	9, 9	47.4
des	16, 16	2.010	16, 16	228.7	2.010	16, 16	227.7	2.010	16, 16	223.8
i8	9, 9	1.400	9, 9	61.2	1.400	9, 9	60.7	1.400	9, 9	60.0
i10	13, 12	3.205	13, 12	163.7	3.191	13, 13	192,9	n/a	n/a	n/a
k2	9, 9	1.528	9, 9	48.2	1.528	9, 9	47.8	1.528	9, 9	47.2
too_large	6, 6	1.380	6, 6	31.2	1.380	6, 6	31.1	1.380	6, 6	30.9

Table V. Experimental Results for 6-LUT Networks

Circuit	array size (N_H, N_V)	$P_{so} = 1\%$ delay (ns)	array size (N_H, N_V)	compile time (s)	$P_{so} = 5\%$ delay (ns)	array size (N_H, N_V)	compile time (s)	$P_{so} = 10\%$ delay (ns)	array size (N_H, N_V)	compile time (s)
c2670	8, 7	1.215	8, 7	234.1	n/a	n/a	n/a	n/a	n/a	n/a
c3540	9, 8	2.220	9, 8	93.2	2.219	12, 12	65.3	n/a	n/a	n/a
c5315	10, 9	1.595	10, 9	286.7	n/a	n/a	n/a	n/a	n/a	n/a
c7552	10, 10	2.624	10, 10	442.4	n/a	n/a	n/a	n/a	n/a	n/a
dalu	9, 8	2.200	9, 8	62.4	2.200	9, 8	61.3	2.200	9, 8	60.5
des	14, 14	1.940	14, 14	337.7	1.940	14, 14	331.7	1.940	14, 14	322.3
i8	8, 8	1.300	8, 8	89.7	1.300	8, 8	90.3	1.300	8, 8	86.6
i10	12, 12	3.426	12, 12	344.6	n/a	n/a	n/a	n/a	n/a	n/a
k2	9, 8	1.480	9, 8	68.8	1.481	9, 8	67.5	1.480	9, 8	66.4
too_large	6, 5	1.200	6, 5	37.4	1.200	6, 5	37.4	1.200	6, 5	37.0

redundant nanowires within NPEs are also important for defect tolerance. Furthermore, as nanowire crossbars become larger, they will contain more defective junctions. Therefore, full utilization of larger nanowire crossbars will require more redundant nanowires.

The necessity of redundant nanowires within NPEs can be analyzed by using extra nanowires and repeating the complete flow again. Table VI targets the same results for 6-LUT implementations as Table V for $P_{so} = 10\%$, but for different redundancy levels. In this table, columns 2-, 4-, and 8-redundant indicate that there are two, four, and eight extra vertical nanowires within the NPEs. $P_{so} = 10\%$ was used to be able to illustrate the effect of redundancy. The results show that all benchmarks were successfully mapped to a minimum-size array by using eight redundant nanowires whereas using two extra nanowires was not enough for some cases. These results demonstrate that usage of redundant nanowires within NPEs can be necessary for successful compilation. If the compilation time of different redundancy levels are compared, it can be seen that it increases slightly with increasing levels of redundancy provided that the benchmark can be mapped onto a minimum-size array successfully by using two extra nanowires. This observation stems from the fact that increasing the level of redundancy increases the number of nodes in the bipartite graph

Table VI. Effect of Redundancy within NPEs for 6-LUT Networks with Defect Rate of $P_{so} = 10\%$

Circuit		2-redundant			4-redundant			8-redundant		
	array size (N_H, N_V)	delay (ns)	array size (N_H, N_V)	compile time (s)	delay (ns)	array size (N_H, N_V)	compile time (s)	delay (ns)	array size (N_H, N_V)	compile time (s)
c2670	8, 7	n/a	n/a	n/a	1.213	8, 7	2305.2	1.222	8, 7	595.8
c3540	9, 8	2.220	9, 8	144.8	2.223	9, 8	160.4	2.229	9, 8	173.5
c5315	10, 9	n/a	n/a	n/a	1.607	10, 9	4094.3	1.605	10, 9	1073.1
c7552	10, 10	n/a	n/a	n/a	2.619	10, 10	7531.0	2.639	10, 10	1720.5
dalu	9, 8	2.203	9, 8	58.0	2.203	9, 8	59.9	2.203	9, 8	66.4
des	14, 14	1.941	14, 14	311.4	1.941	14, 14	322.2	1.941	14, 14	354.3
i8	8, 8	1.302	8, 8	83.5	1.302	8, 8	87.4	1.302	8, 8	94.8
i10	12, 12	3.440	13, 13	652.8	3.419	12, 12	3500.7	3.431	12, 12	1018.2
k2	9, 8	1.481	9, 8	68.5	1.481	9, 8	71.0	1.481	9, 8	77.6
too_large	6, 5	1.203	6, 5	36.3	1.203	6, 5	37.2	1.203	6, 5	38.4

and therefore increases the runtime of the heuristics. On the other hand, if two extra nanowires are not enough for successful mapping, then the compilation time reduces drastically with increasing levels of redundancy. In this case, the increase in the number of nodes helps find a solution faster, which in turn helps the runtime go down. One final remark can be made on critical path delays. The results show that it is not affected much by changing the redundancy level.

Another way to interpret the simulation results is to compare the results of the same benchmark over different LUT implementations. The first comparison can be made for critical path delays. The results show that larger the LUTs used in the implementation, smaller the critical path delay. Although larger LUTs require more time to produce the output, this result is due to the fact that there is smaller inter-cluster communication, which is the major contributor to critical path delay.

The other criterion is the array size in terms of the number of hybrid clusters required. Since implementations with larger LUTs require fewer clusters, the array size decreases with an increasing number of LUT inputs. However, the overall physical area of the chip increases due to the exponential increase in NPE area with an increase in the number of LUT inputs.

The last comparison can be made based on the compile time. Although increasing the number of inputs to LUTs causes a decrease in the number of clusters, the compilation time typically increases. This stems from the fact that the heuristic used to decide on whether the mapping is valid takes more time for the larger NPEs. Since the number of vertical nanowires within each NPE and the number of product terms to be mapped increases exponentially, the runtime of the heuristic increases and a decrease in the number of clusters is not enough to decrease the overall runtime.

6. CONCLUSION

In this article, we proposed and evaluated a fault-tolerant architecture based on nanowire crossbar-style processing elements. The aim of this architecture was to target errors caused by not only permanent defects, which may or may not be located during testing, but also transient faults that occur at runtime.

This goal was achieved by (i) using hierarchical, multilevel redundancy based on replicating nanowires within an NPE, as well as replicating the NPEs themselves, and (ii) employing CMOS voting logic to correct errors produced by the NPEs.

We also described a compiler for this architecture. We proposed fast heuristics to map the logic to defective, redundant components. We also proposed an annealing-based placer that uses the defect information within its cost function. By restricting defect-awareness to the back-end, we increase the retargetability of the compiler across different defect models.

REFERENCES

BETZ, V. AND ROSE, J. 1997. VPR: A New Mapping, Placement and Routing Tool for FPGA Research. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*. 213–222.

BROWN, J. AND BLANTON, R. 2004. CAEN-BIST: Testing the nanofabric. In *Proceedings of the International Test Conference* 462–471.

DEHON, A. 2005. Nanowire-based programmable architectures. *ACM J. Emerg. Techn. Comput. Syst. 1*, 109–162.

DEHON, A. AND NAEIMI, H. 2005. Seven strategies for tolerating highly defective fabrication. *IEEE Des. Test Comput.' 22*, 306–315.

DUAN, X., HUANG, Y., AND LIEBER, C. 2002. Nonvolatile memory and programmable logic from molecule-gated nanowires. *Nano Lett. 2*, 487–490.

HUANG, Y., DUAN, X., CUI, Y., LAUHON, L. J., KIM, K., AND LIEBER, C. M. 2001. Logic gates and computation from assembled nanowire building blocks. *Science 294*, 1313–1317.

ITRS. 2005. International Technology Roadmap for Semiconductors. http://public.itrs.net.

JACOME, M., HE, C., DE VECIANA, G., AND BIJANSKY, S. 2004. Defect tolerant probabilistic design paradigm for nanotechnologies. In *Proceedings of the Design Automation Conference* 596–601.

MISHRA, M. AND GOLDSTEIN, S. 2003. Defect tolerance at the end of the roadmap. In *Proceedings of the International Test Conference* 1201–1210.

MITRA, S., SAXENA, N., AND MCCLUSKEY, E. 2000. Common-mode failures in redundant VLSI systems: A survey. *IEEE Trans. Reliab. 49*, 285–295.

MUKHERJEE, S., EMER, J., AND REINHARDT, S. 2005. The soft error problem: An architectural perspective. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 243–247.

PTM. 2007. Predictive technology model. http://www.eas.asu.edu/~ptm.

RAD, R. AND TEHRANIPOOR, M. 2006. A new hybrid FPGA with nanoscale clusters and CMOS routing. In *Proceedings of the Design Automation Conference* 727–730.

RAO, W., ORAILOGLU, A., AND KARRI, R. 2006. Topology aware mapping of logic functions onto nanowire-based crossbar architectures. In *Proceedings of the Design Automation Conference* 723–726.

REJIMON, T. AND BHANJA, S. 2006. Probabilistic error model for unreliable nano-logic gates. In *Proceedings of the IEEE Conference on Nanotech*. 47–50.

SCHMIT, H. AND CHANDRA, V. 2005. Layout techniques for FPGA switch blocks. *IEEE Trans. VLSI. Sys. 13*, 96–105.

SENTOVICH, E. M., SINGH, K. J., LAVAGNO, L., MOON, C., MURGAI, R., SALDANHA, A., SAVOJ, H., STEPHAN, P. R., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. 1992. SIS: A system for sequential circuit synthesis. Tech. rep. UCB/ERL M92/41. Department of Electrical Engineering, University of California, Berkeley.

SHIVAKUMAR, P., KISTLER, M., KECKLER, S., BURGER, D., AND ALVISI, L. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference Dependable Systems and Networks*. 389–398.

SNIDER, G., KUEKES, P., HOGG, T., AND WILLIAMS, R. 2005. Nanoelectronic architectures. *Appl. Phys. A 80*, 1183–1195.

SNIDER, G. AND WILLIAMS, R. 2007. Nano/CMOS architectures using a field-programmable nanowire interconnect. *Nanotech. 18*.

STRUKOV, D. AND LIKHAREV, K. 2005. CMOL FPGA: A reconfigurable architecture for hybrid digital circuits with two-terminal nanodevices. *Nanotech. 16*, 888–900.

TAHOORI, M. 2005. A mapping algorithm for defect-tolerance of reconfigurable nano-architectures. In *Proceedings of the International Conference on Computer-Aided Design*. 668–672.

ZIEGLER, M. AND STAN, M. 2003. The CMOS/nano interface from a circuits perspective. In *Proceedings of the International Symposium Circuits and Systems*. 904–907.

Received July 2008; revised December 2008; accepted March 2009 by Fabrizio Lombardi