Defect Tolerant Probabilistic Design Paradigm for Nanotechnologies[*]

Margarida Jacome, Chen He, Gustavo de Veciana, and Stephen Bijansky
Department of Electrical and Computer Engineering
The University of Texas at Austin
{jacome,che,gustavo,bijansky}@ece.utexas.edu

ABSTRACT

Recent successes in the development and self-assembly of nanoelectronic devices suggest that the ability to manufacture dense nanofabrics is on the near horizon. However, the tremendous increase in device density of nanoelectronics will be accompanied by a substantial increase in hard and soft faults, posing a major challenge to current design methodologies and tools. In this paper we propose a novel probabilistic design paradigm for defective but reconfigurable nanofabrics. The new design goal is to devise an appropriate structural/behavioral decomposition which improves scalability by constraining the reconfiguration process, while meeting a desired probability of successful instantiation, i.e, yield. Our approach not only addresses the scalability problem in configuring dense nanofabrics subject to defects, but gives a rich framework in which critical trade-offs among performance, yield, and per chip cost can be explored. We present a concrete instance of the approach and show extensive experimental results supporting these claims.

Categories and Subject Descriptors: J6 [Computer-Aided Engineering]: Computer-aided design

General Terms: Performance, Design, Reliability.

Keywords: Nanotechnologies, probabilistic design, defect tolerance.

1. INTRODUCTION

Fast paced progress on devising and assembling nanoelectronic devices suggests that it will be possible to manufacture large-scale computation nanofabrics within 10-15 years [1, 2, 3, 4, 5, 6, 7]. Aside from greatly increasing concerns with complexity and scalability, nanoelectronic fabrics share a characteristic that will impact the entire design hierarchy. Irrespective of the 'winning' technologies, e.g., semiconductor nanowires and/or carbon nanotubes, it is widely recognized that devices and interconnect at the nanoscale will exhibit fault densities much higher than state-of-the-art silicon technology. Indeed, they will have: (1) a density of defects which is much higher than current silicon technologies [8, 9, 3, 5]; and (2) are likely to be much more susceptible to transient faults, sometimes referred to as soft faults, see e.g.,[10, 3]. The increases are, in part, due to the physical dimensions being considered. Indeed, from a materials perspective, decreasing the size of structures increases the ratio of surface area to volume, making imperfections on surfaces or materials' boundaries more critical to proper function of nanoscale interconnects and devices. Furthermore, at such reduced scales, the discrete nature of atomic matter and charge becomes significant. Namely, a single charge or defect may significantly impact the structural stability of a nanodevice as well as its sensitivity to the electrostatic environment. These observations point to a reliability problem that is intrinsic to nanoscale regimes, and is thus here to stay.

Overcoming this problem will require work at many levels, including devising more reliable devices, interconnects, manufacturing processes, and materials. At the same time, one needs to start devising design principles, abstractions, and tools to enable system level designers to address the projected increases in faults. These will not only be pivotal to conclusively demonstrating the viability of nanotechnologies, but also critical to moving them from labs to production. Current design methodologies and tools take high reliability for 'granted', and thus, their direct application to designing nanosystems would lead to exceedingly low yields [5]. A paradigm shift in design methods and tools is thus required, placing defect and fault-tolerance at the forefront, and recognizing these as inextricably tied to system performance. Moreover given the substantial degree of uncertainty inherent to nanoscale technologies, design will have to be framed in a probabilistic setting, if effective optimization of performance, yield, and other key figures of merit is to be achieved.

It has been demonstrated that nanoelectronics technology is well suited to building reconfigurable computational fabrics[1, 9, 11]. This is significant because reconfigurability provides a powerful tool to circumvent the uncertainty associated with distributions of defects. However, designing complex systems to be instantiated through reconfiguration poses a major scalability challenge – defect mapping and configuration must be performed on a per chip basis. This paper proposes a novel probabilistic design paradigm targeting reconfigurable architected nanofabrics and shows that it lays a promising foundation towards comprehensively addressing, at the system level, the density and reliability challenges posed by emerging nanotechnologies. Indeed, first it provides a hierarchy of design abstractions aimed at ensuring scalability, not only during a nanosystem's synthesis, but also in the defect mapping and configuration phases. Scalability must be jointly addressed across these phases – this is one of the innovative aspects of our approach. Second, the proposed hierarchy is based on abstractions that enable an effective integration of fault-tolerance and defect-avoidance techniques in design methodologies. We will show that this is key to enabling the design of robust nanosystems. A third innovation embodied in our approach is that it provides an adequate probabilistic framework in which to consider critical system-level design trade-offs between performance, yield and per chip costs.

The paper is organized as follows. In §2 we briefly overview previous relevant work on fault-tolerant design and defect avoidance. Our defect tolerant probabilistic design paradigm is introduced in §3. Experimental results demonstrating the promise of the approach are discussed in §4. Finally, §5 concludes with a brief discussion of how our approach can be used to address both the defect-avoidance and fault-tolerance challenges posed by emerging nanotechnologies.

2. PREVIOUS WORK

Classical fault tolerance approaches. Triple-module-redundancy and N-module-redundancy (TMR and NMR, respectively) are based on the use of multiple modules and reliable arbitration units, see e.g., [12]. Basing a design methodology on TMR/NMR would mean allocating a priori structural redundancy and arbitration resources such that defects could be tolerated with high probability in fabricated chips.

[*]This work is supported in part by SRC Grant CRS 1152.001 and NSF Grant CCR 0310119.

However, the reliability of such designs is limited by that of the final arbitration unit, making the approach difficult in the context of highly integrated nanosystems. Nevertheless, in the sequel we use a TMR-design methodology as a baseline for comparison with our approach.

The 'arbiter reliability bottleneck' problem could be circumvented through coding. Unfortunately, efficient codes to achieve reliable *computation* are either exceedingly complex or limited to linear operations, see e.g.,[13, 14, 15]. The challenge lies in the transformational character of computation. Even with the high densities afforded by nanotechnologies, the resource/interconnect complexity to realize coded computation is impractical.

Defect avoidance approaches. An approach to nanosystem design is proposed in [9, 11], which consists of first mapping defects on a large reconfigurable regular grid of *nanoblocks*, each of which can be configured as an AND, OR, XOR, half-adder etc., then synthesizing (offline) a feasible configuration realizing the application for each nanofabric instance, and finally configuring each instance accordingly. This approach has its roots in the TERAMAC experiment[16, 8]. Unfortunately, an *unstructured* approach that requires mapping, synthesis and configuration at such a fine granularity would not scale for large nanosystems. In addition, the group testing strategy used for defect mapping in [9, 11] requires unlimited connectivity among nanoblocks. Still, the key idea of using reconfiguration to achieve defect avoidance is the starting point for our work.

3. PROBABILISTIC DESIGN PARADIGM

Our target implementation platform is a (re)configurable nanofabric. Our approach is based on two key ideas. The first is to structure designs as hierarchies of carefully dimensioned (re)configurable fabric regions while decomposing and assigning functional 'flows' to each region. By restricting the functionality which is preassigned to a specific nanofabric region, we limit the scope and complexity of the defect mapping and configuration tasks that need to be performed for each chip. Namely, since one need only work with a set of basic flows assigned to limited complexity and structured fabric regions, it is possible to precompute configuration alternatives. However, to achieve high yields, one must ensure up-front that each region has sufficient degrees of freedom for configuration (i.e., 'capacity'), so that the associated flows can be instantiated with high probability.

The second idea underlying our approach is to devise efficient defect mapping and configuration methods for such regions. Observe that one need not map all defects in a region, but instead, only establish the existence of a feasible configuration for the region's associated flow(s). Also observe that the defect mapping process is time consuming and one might wish to compromise its accuracy, i.e., allow false negatives, in order to trade-off accelerated fabrication with lower yield. Our approach captures these trade-offs, providing a good foundation for design methodologies for unreliable nanotechnologies.

3.1 Proposed Abstractions and Design Hierarchy

The nanofabric is architected using the three-level hierarchy summarized in Fig.1. *Regions* are the basic configuration units, i.e., the structural 'primitives' of the nanofabric, while *basic flows* are the corresponding behavioral primitives. One may think of basic flows as instructions and regions as the programmable execution units of a nanocomputing system. They are defined such that each flow can be instantiated (with high probability) into a single region if it has a 'moderate' number of faulty elements. To be concrete, in this paper we will assume each region contains eight processing elements (PEs), each capable of performing the standard set of 8-bit arithmetic/logic operations – such small PEs can be realized using simple designs, e.g., memory based, and incorporate redundancy. Thus, basic flows may contain at most eight operations. Fig.1 shows the 7 basic flow types we will use, ft1–ft7. Naturally, the larger the number of alternative ways a flow can be instantiated in a given region, the larger the probability of a successful configuration when defective elements are present.

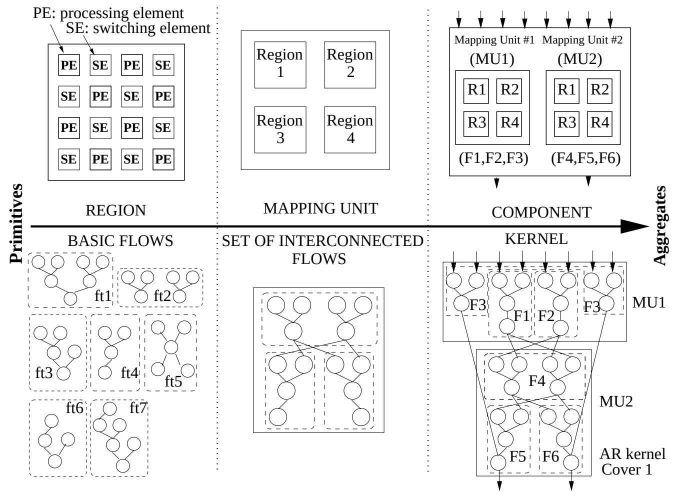

Figure 1: Three level design hierarchy.

The second level of the design hierarchy comprises *mapping units* (MUs), each aimed at realizing a (connected) set of flows. Mapping units are dimensioned so as to provide sufficient redundant resources, i.e., configuration 'capacity', to ensure their associated flows can be instantiated with the budgeted probability. Fig.1 illustrates this, exhibiting a mapping unit requiring a minimum of three regions (one for each of its associated basic flows), and yet containing four such regions, so as to increase the probability of a successful configuration. Mapping units also define the scope in which to consider alternative configurations for a given behavior, i.e., set of flows. Specifically, one need only determine whether the flows assigned to each mapping unit can be realized on its *internal* regions.

The top level of our structural design hierarchy is the *component* abstraction. Each component is intended to either realize a *transformational kernel*, i.e., perform an application's computation and control tasks, or realize *storage and switching resources*, i.e., support communication between tasks. This paper focuses on the first type of component, as their transformational nature is more challenging from a reliability standpoint. Each application kernel is thus mapped to a corresponding component or set of component 'slices', depending on the required word size. At the right most side of Fig.1 we illustrate one such component, implementing an auto-regression (AR) filter kernel. The component design includes two mapping units (MU1 and MU2), each with four regions – flows F1, F2, and F3 of the AR kernel are assigned to MU1, while flows F4, F5, and F6 are assigned to MU2. In order to simplify scheduling and control, the set of flows assigned to the various mapping units of a component must satisfy convexity constraints, that is, there cannot be 'circular' (input/output) data dependencies among them, see e.g.,[17]. In order to control intra-MU routing complexity (see below), our nanofabric architecture allows mapping units to have at most nine regions.

Routing among elements in the hierarchy is supported as follows. Within a region switching elements (SEs) are used to route between adjacent PEs. We currently assume each SE can support up to two routing channels among its adjacent PEs. At the MU level, each region is surrounded by a routing track on each of its sides. A switch block is placed between each routing track. Since there are a relatively small number of tracks within an MU, and MUs may contain at most nine internal regions, efficient table lookup based algorithms can be used to explicitly program the switch blocks with the shortest paths between any two regions. At the component level, inter-MU routing is done using long-lines for signals that are run next to the MUs and switch boxes for routing between MUs. The routing strategy is similar to that used for intra-MU routing.

To summarize, the *component* abstraction implements an interface that hides/encapsulates the particular defect realizations for a nanofabric instance – that is, all operational (i.e., successfully configured) systems are structurally 'identical' at the component level. Note however that there still is uncertainty associated with the actual *performance* of such components. Namely, there will be delay variability across different component instances, and components are still susceptible to transient/soft faults, and may thus malfunction. Still, the component abstraction provides a basic foundation towards controlling the complexity associated with handling the remaining sources of uncertainty, see §5.

Components and their constituent elements decompose the three main problems that need to be handled – namely, the *design* of a suitable configurable nanofabric architecture for a target nanosystem, the *mapping of defects*, and the *configuration* of actual nanofabric instances – into smaller subproblems which can be handled quasi-independently. Together these make our approach inherently scalable. Moreover, because each subproblem is small and relatively simple, solutions can be explicitly 'enumerated'. Furthermore the internal computation capacity of each chip can be effectively used to assist in its own defect mapping and configuration, see §3.2.

3.2 Scalable Defect Mapping and Configuration

A strength of our approach is that it enables a substantial part of the defect mapping and configuration tasks for structured nanofabrics to be performed within the nanofabric itself. Specifically, a region's defective components and/or connectivity are systematically identified by a suite of prespecified *test tiles*. A test tile corresponds to configuring and operating a set of PEs to perform a function whose output allows detection of possible defects. We propose using tiles implementing a triple-module-redundancy (TMR) configuration. Specifically each tile includes four processing elements, where one plays the role of an arbiter for the outputs of the other three. Such small tiles can be configured systematically, and will usually permit locating faulty PEs or connections. Specifically, a TMR testing tile can identify faulty PE/connectivity if the arbiter and two other PEs/connections are operational.[1] Fig.2 shows four such tiles with the arbiters labeled 'A.' The results produced by the four tiles shown in the figure can, for example, detect if flow $ft2$ (see Fig.1), can be configured in a region, even if, say, the two processing elements labeled 'F' in Fig.2(a) are faulty, see Fig.2(b).

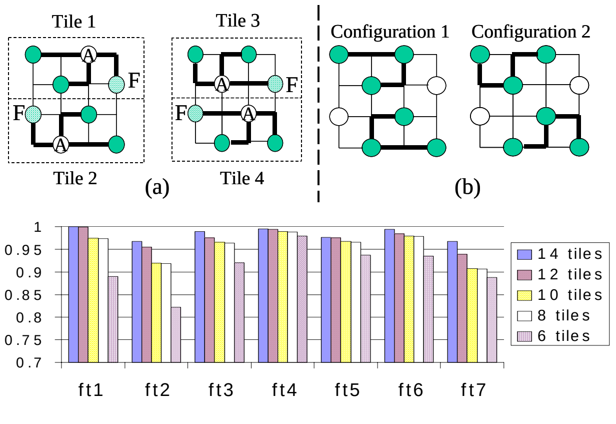

Figure 2: Example showing (a) four TMR testing tiles and (b) two feasible configurations for basic flow $ft2$. The graph shows coverage for each basic flow when tile testing suites consisting of 6–14 TMR tiles are used and $(P_e, P_a, P_c) = (10, 5, 1)\%$.

The region size and topology adopted in our current fabric architecture were selected to realize a good trade-off between: (1) the amount of raw redundant capacity provided at the region level; and (2) the number of TMR testing tiles required to achieve a good flow-oriented defect coverage. Specifically, we have experimentally determined that, if the probability of failure for processing elements P_e is 1-20%, for connections P_c is 0.1-2%, and for processing elements operated as arbiters P_a is 0.5-10%, then 14 TMR tiles provide an excellent defect coverage of a region for our set of basic flows. Note that one would expect $P_a < P_e$ since a PE's operation as an arbiter is a subset of the arithmetic/logic operations supported by a PE as a generic processor. The use of fewer tiles results in a degradation in coverage, and thus increases the number of false negatives. Fig.2 shows the coverage, i.e., one minus probability of false negatives, for each basic flow, achieved by testing suites including 6–14 tiles. Clearly, the more tiles are used, the better the defect map one obtains, i.e., the better the coverage. Yet a large number of tiles will mean a higher cost, in time, for producing a chip.

[1]We assume that a PE configured as an arbiter which is faulty is very unlikely to generate a false positive, i.e., generate a 'correct' diagnosis message.

We conclude by noting that our approach shares some commonalities with previous work in high level synthesis, see e.g.,[17]. However, the performance versus yield trade-offs exposed in our framework add a *new design/optimization dimension* that will be critical for unreliable nanotechnologies. Scalability issues associated with configuring ultra large/dense nanofabrics to circumvent defects pose yet another 'uncharted' challenge.

4. EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Empirical Validation Methodology

We evaluated the proposed design approach by considering a number of possible *component designs* for the benchmark kernels shown in Table 1. A component design for a kernel requires defining the following:

(1) Sizing the component's resources, i.e., specifying the number of MUs and number of regions within each MU;
(2) Specifying a flow cover for the kernel, using a set of basic flows, see examples in Fig. 4;
(3) Specifying an assignment of basic flows to MUs, whereby basic flows in a cover are assigned to MUs subject to convexity constraints.

Kernels	#Ops	#Ops in CP	#Covers	Granularity of covers
FIR Filter unrolled (FIRu)	32	11	2	4/6
Auto-Regression Filter (AR)	28	8	2	6/7
Avenhous Filter mod (AF2)	19	7	2	6/7
Avenhous Filter (AF1)	18	7	2	6/6
2D-DCT (DCT)	16	9	2	4/6
FIR Filter (FIR)	16	9	2	4/6
FFT	10	3	1	5

Table 1: Characteristics of benchmark kernels.

We assessed the merits of each component design when defects are present via Monte Carlo simulation. Different regimes were parameterized by probabilities of failure for processing elements (PEs), PEs operating as arbiters, and switches/connections, P_e, P_a and P_c respectively. We focus on defects in the *transformational* parts of a component (i.e., regions) as these are the most challenging to handle. Specifically, we applied a suite of 14 TMR testing tiles to each region of a component instance, to obtain a (partial) defect map. Then for each region inside an MU, a table lookup based algorithm was used to identify which flows associated with the MU are feasible on that region. Finally, a heuristic algorithm was used to decide where to instantiate each basic flow. The algorithm considers flows assigned to a given MU in topological order, and regions within the MU in row first order, and greedily configures flows on the first available feasible region. This simple heuristic tries to minimize inter and intra-MU delays. If this is not successful, we then attempt to map flows based on their size, with coarser/harder flows first. Naturally, this heuristic may fail to find a feasible configuration– because none exists, the defect map is incomplete, or the heuristic did not find it. For a given defect regime, (P_e, P_a, P_c), we sampled a large number of defect realizations, to estimate the probability that basic flows will be feasible on a region, probability of false negatives, and average delays, with adequate confidence intervals. The probability that a component design fails to be configured is *computed* based on the above estimates and particulars of the design.

We modeled delays associated with a given component instance as follows. Each operation takes 2 cycles to complete on a PE and have its results routed to an adjacent PE, i.e., to a consumer operation of the same flow. Thus, the delay (in number of cycles) of any basic flow on a region will be the number of operations on its critical path times two. When a signal leaves a region, it takes 1 cycle to traverse a side of a region and go through a switchbox to an adjacent region. Similarly, a signal produced in one MU and consumed by another MU takes 1 cycle to traverse the length of a region through a long-line. Operations mapped to the same MU can start executing as soon as their operands become available. However, in order to simplify control within components, we imposed an additional scheduling constraint for flows belonging to different MUs. Specifically, an MU can begin execution only after all of its 'producer' MUs (i.e., MUs that generate inputs to it) have completed execution. Finally, note that the relative delays for computation versus transport is what truly matters here. Although we are only able to present one scenario in this paper, one can roughly infer the impact of changing the relative values.

We let P_f denote the probability that a component *fails* to be configured. In order to fairly compare the relative performance, i.e., delay, of alternative component designs, we only compare designs that achieve the *same* P_f. Let CP denote the critical path length of a kernel. Then, under our delay model, $CP_{delay} = 2 \times CP$ is a lower bound on the delay for any component realizing that kernel. We will let RP denote the *relative performance* of a component, defined as CP_{delay} over the actual delay achieved by the component. A lower RP implies a higher delay overhead due to data transfers across regions and MUs. Since the delay of a component design will vary depending on the realization of defects, in the sequel we will let RP_{avg} and RP_{wc} denote ratios of CP_{delay} over the average and worst case delay, over a sample of instances that were simulated. To compare the performance of alternative component designs over a range of P_f's, we define the *normalized relative performance* $NPR(P_f)$ of a design as its performance, say RP_{avg}, divided by the performance of the best design considered, irrespective of P_f. Thus, a graph of $NPR(P_f)$ exhibits the relative performance penalty of various design alternatives, as one varies P_f.

4.2 Contrast to TMR-based Design Methodology

We first contrasted the effectiveness of our approach versus a design methodology based on triple-module-redundancy (TMR), see §2. By a TMR-based design, we mean designs based on a priori allocation of redundant resources and arbiters, so that a target P_f is met. This approach requires a single synthesis step and no defect mapping or configuration. Below we discuss results for the FFT kernel, see Table 1. Two defect regimes (technologies) were considered $(P_e, P_c) = (10, 1)\%$ and $(P_e, P_c) = (1, 0.1)\%$, see Fig.3. For each target P_f, we generated two TMR-based designs: one assuming perfect arbiters, i.e., P_a = 0 (unrealistic for nanotechnologies); and the other assuming $P_a = P_e/1000$. We assumed optimistically that data transfer delays across regions and MUs for TMR-based designs would be zero. For component designs based on our approach, we assumed pessimistically that $P_a = P_e/2$. The FFT kernel can be covered by two basic flows of type $ft5$ (see Fig.1), and thus we considered only component designs that assign both flows to a single MU. As P_f decreases, our component designs include an increasing number of regions within the MU, so as to meet the target P_f.

Fig.3 exhibits $NPR(P_f)$ using RP_{wc} for our approach and RP for TMR-based component designs – note that there is no delay variability in the latter. Recall that $NPR(P_f)$ is normalized to the relative performance for the best performing design over all P_f considered. As expected, TMR-based designs with non-ideal arbiters eventually reach an arbitration bottleneck beyond which they can no longer reduce P_f. Also note that (*unrealistic*) TMR-based designs with perfect arbiters

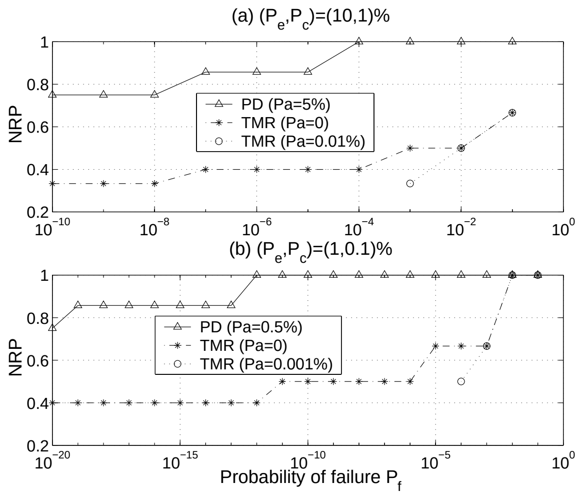

Figure 3: Normalized relative performance for probabilistic designs (PD) and TMR-based based designs for the FFT kernel.

can achieve P_f targets, yet exhibit a much longer delay than the *worst case* for our probabilistic component designs. This is due to the large amount of a priori redundancy required to achieve a small target P_f. Thus, from the perspective of creating defect tolerant designs, TMR is inappropriate, i.e., delivers low yield and performance. Yet, a TMR-based design may at the same time be robust to soft/transient faults. We shall return to this point in §5.

4.3 Exposing Performance versus Yield Trade-offs

In this section we provide experimental evidence to support the claim that our design approach and hierarchy expose a new class of trade-offs that will be critical for designing computing systems on defect prone nanotechnologies.

4.3.1 Basic Illustrative Examples

We illustrate these trade-offs by analyzing results for component designs for the DCT and AR kernels, see Table 1. For each kernel, two different flow covers are considered: those for the AR filter are shown in Fig.1 and 4, and those for the DCT kernel are shown in Fig.4. For both kernels, Cover 2 uses larger basic flows than Cover 1. For each cover, three different assignments were considered: (1) assign all basic flows to one MU; (2) assign half of the basic flows to one MU and the rest to another MU; and (3) assign each basic flow to its own MU. Thus, for each kernel, we considered 6 different basic component designs, i.e. 2 covers with 3 assignments each. The following naming convention is used in the sequel: $KERNELx.y$ will denote a design for a $KERNEL$ where cover x and assignment y were used. For instance, $DCT1.2$ represents a component design where F1, F2 are assigned to MU1 and F3, F4 are assigned to MU2. Finally for each design, we varied the number of regions on each MU, so as to meet the target P_f.

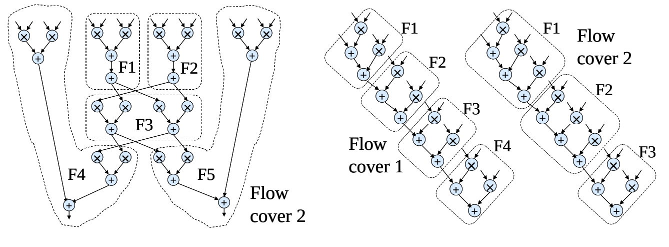

Figure 4: On the left Cover 2 for AR filter kernel and on the right Covers 1 and 2 for 2D-DCT kernel.

Fig.5 and 6 exhibit $NRP(P_f)$ for the six AR and DCT component designs under two defect scenarios: Case 1 where $(P_e, P_a, P_c) = (10, 5, 1)\%$; and, Case 2 where $(P_e, P_a, P_c) = (1, 0.5, 0.1)\%$. Here,

$NRP(P_f)$ for given component design meeting a target P_f corresponds to the ratio of the *average delay* for the best design (over all P_f) to that of the component design under consideration. Consider Case 1 which has a higher density of defects. Note that the designs using covers with larger flows and a single MU (i.e., AR2.1 and DCT2.1) deliver the best performance but are also the most constrained in terms of yield they can achieve. Indeed, the use of coarser flows reduces the number of inter-flow edges, thus reducing delay overheads associated with communication across regions. Furthermore the use of a single MU eliminates overheads associated with data transfers across MUs. However, coarser flows are harder to successfully configure on a defective region, and the use of a single MU limits drastically the redundancy one can add to compensate for the problem (recall that an MU can contain at most 9 regions). The designs with the next best performance are those using a single MU, but covers with smaller flows, i.e., AR1.1 and DCT1.1. This suggests that the number of mapping units in a design has a substantial impact on average delay. However, in order to achieve lower P_f, one must consider component designs with more mapping units, and pay the performance penalty. The next best designs are those with two mapping units and coarser covers. Finally, as might be expected, component designs that can achieve the lowest P_f in both cases are the ones that use covers with finer-grained flows and allocate one MU per flow. The results for Case 2, where the defects densities are 10 times lower, exhibit the same basic trends, except that lower delay overheads are seen for the same P_f values, since less region redundancy is required. Furthermore the best achievable P_f for all designs decreases/improves substantially.

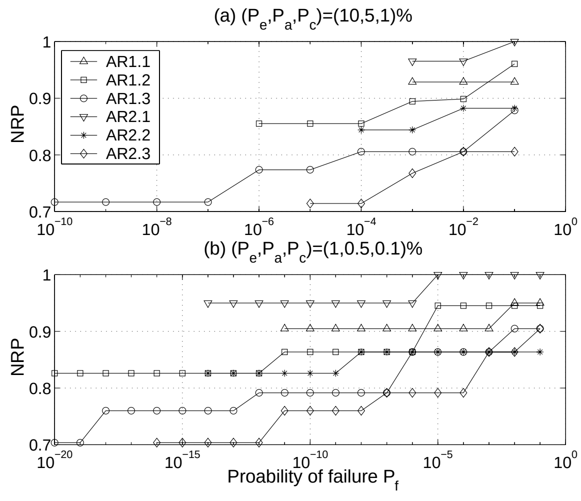

Figure 5: Normalized relative performance (NRP) for various component designs for AR kernel.

A fundamental trade-off between the probability of successful configuration (i.e., yield) and delay emerges from these experiments. A summary analysis suggests that, if the probability of unsuccessfully configuring a component (P_f) is not sufficiently low, one should consider three options: (1) instantiating more regions in the mapping units; (2) using a cover with smaller flows; and (3) instantiating more mapping units and assigning fewer flows to each mapping unit. Our experiments indicate that, as we progress from Option 1 to 3, the achievable P_f decreases sharply, while the average delay increases. Alternatively, one might choose a higher P_f and thus decrease yield in order to improve the average delay of the resulting component instances. These preliminary considerations exhibit the effectiveness of our approach in exposing yield, delay, and cost trade-offs.

Finally, we note that a design methodology based on reconfiguration to circumvent defects will need to contend with performance variability. For example, Fig.7 shows the relative performance RP, i.e., our lower bound CP_{delay} divided by the average, best and worst case delays of the best AR component design (in terms of average delay) for each P_f. As it can be seen, the best and worst case relative performance are within 10–15% of the average – we shall return to this important point in §5.

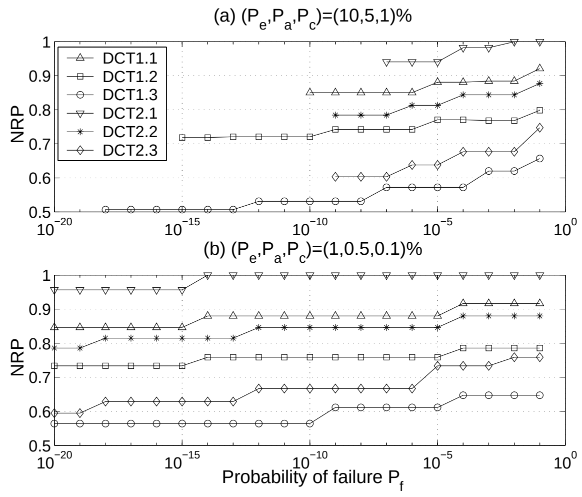

Figure 6: Normalized relative performance (NRP) for various component designs for 2D-DCT kernel.

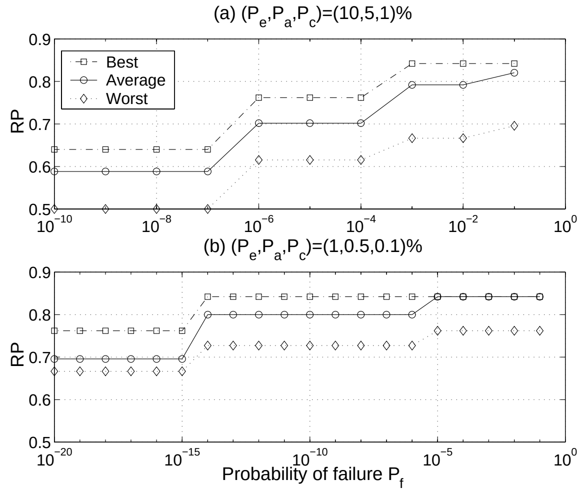

Figure 7: Best,worst and average relative performance (RP) of optimal designs for AR kernel.

4.3.2 Trend Analysis

In this section we provide further empirical results towards identifying the key design issues that need to be considered during design space exploration for each component. To that end, we will consider the first six kernels in Table 1. As before, two covers were devised for each kernel – Table 1 indicates the size of the largest flow in each cover. We consider the same three basic alternative component designs discussed in §4.3.1, so again six basic component designs per kernel. In order to allow comparisons across component designs associated with *different* kernels, we will normalize the resulting delays by the length of the corresponding critical path, thus obtaining an effective delay *per operation* on the critical path. The normalized relative performance per operation NRP^o of a design is the average delay divided by its CP_{delay} normalized to the best such ratio obtained over all designs and P_f considered. Further we note that a kernel with more operations is likely to see higher delay overheads to meet the same overall P_f. Thus, we define a normalized probability of *operation failure* P_f^o achieved by a component design associated with a kernel with N operations as follows:

$$P_f^o = 1 - (1 - P_f)^{(1/N)}.$$

Note that $1 - P_f^o$ can be interpreted as geometric average of the probability of success in realizing each operation (where they are assumed independent) achieved by a component design meeting a target P_f.

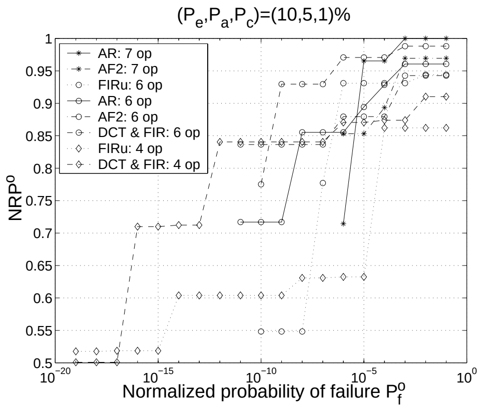

Figure 8: Impact of cover granularity.

Let us consider the impact of cover granularity, defined as the size of the largest constituent basic flow, on the average delay and P_f. To do so, we will restrict our attention to kernels for which two covers with distinct maximum size flows have been defined, see Table 1. For each kernel, cover and P_f, we determined which among the three design alternatives (see §4.3.1) minimizes the average delay, i.e., has maximum NRP^o. Thus, the reported performance for each P_f corresponds to the best component design for the particular cover. Fig. 8 exhibits normalized performance NRP^o versus the normalized probability of failure P_f^o for the kernels considered. The kernel and cover granularity are indicated in the labels. As expected, the covers with the smaller flows deliver worse delay performance NRP^o but higher normalized probability of failure P_f^o. The three major drops in performance for each cover correspond to increases in the number of MUs and corresponding changes in basic flow assignments, to ensure the target P_f, while smaller drops correspond to increases in regions per MU. To summarize, this experiment shows that, when several alternative component designs (using different granularity covers) achieve the same P_f, the best (i.e., minimum average delay) designs will be those relying on the coarser covers. Yet, not all covers with the same granularity are equally good. The following set of experiments illustrates this point.

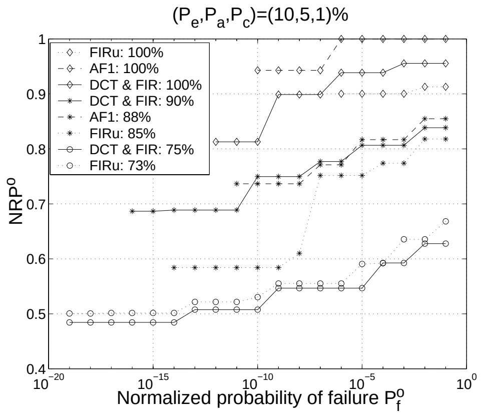

Figure 9: Impact of exposed instruction level parallelism.

As mentioned earlier, to simplify control within components, we assumed that MUs start execution only after all of their 'producer' MUs have completed. Thus, for some component designs, parallelism may be lost, in the sense that operations that could execute concurrently are serialized. Accordingly, in general, the larger the number of MUs in a design, the higher the potential for lowering the instruction level parallelism (ILP) in a kernel. Note that the AR kernel is not included in this experiment since, for the two considered covers, none of three MU allocation/assignment alternatives would reduce ILP. Fig.9 exhibits results obtained for a number of alternative component designs. As seen in the figure, the normalized performance of the designs clusters into three groups based on the exposed ILP: 100%, 90–85%, and 75–73%. For each cluster, NRP^o is impacted by other factors, including the size of the kernel, number of interflow edges, etc., yet the amount of exposed ILP is clearly a key factor. These results suggest that design methods will need to select covers and MU assignments preserving ILP. Note that the component designs with more MUs are most likely to decrease ILP, yet they are also most capable of delivering very low P_f, or normalized P_f^o, e.g., DCT/FIR 75% and FIRu 73%. So, once again, the recurring yield-performance trade-off emerges.

5. CONCLUSIONS AND ONGOING WORK

As mentioned in the introduction, nanotechnologies are also likely to have a higher susceptibility to transient/soft faults. This problem can be addressed by adding structural redundancy at the region, mapping unit, and/or component levels. Unfortunately, this is likely to once again lead to performance degradation. In ongoing work, we are using the component abstraction proposed in this paper to enable the design of *high performance* robust nanosystems. Specfically, we have devised a promising technique, reliability-driven speculation, whereby computations/kernels are speculatively executed on faster but less reliable components, with on-chip verification (on slower but more reliable components) and roll back support. Our preliminary results show that, if speculation is based on fairly reliable components, the overall system throughput is dominated by the faster 'speculative' components rather than by the slower components necessary for verification. In addition, we are investigating the use of simple load balancing techniques (implemented on replicated speculative components) so as to minimize performance variability across nanochip instances. These ideas are still preliminary, but illustrate the potential of the foundation proposed in this paper.

6. REFERENCES

[1] C.P. Collier et. al., "Electronically configurable molecular-based logic gates," *Science*, vol. 285, pp. 391–94, July 1999.

[2] A. Bachtold et al, "Logic circuits with carbon nanotube transistors," *Science*, vol. 294, pp. 1317–20, 2001.

[3] T. I. Kamins and R. S. Williams, "Trends in nanotechnology: Self-assembly and defect tolerance," in *Proc. NSF Partnership in Nanotechnology Conf.*, Jan. 2001.

[4] Y. Huang et al., "Logic gates and computation from assembled nanowire building blocks," *Science*, vol. 294, no. 5545, pp. 1313–7, 2002.

[5] G. Bourianoff, "The future of nanocomputing," *Computer Magazine*, pp. 44–49, Aug. 2003.

[6] A. DeHon, "Array-based architecture for FET-based nanoscale electronics," *IEEE Trans. Nanotechnology*, vol. 2, no. 1, pp. 23–32, 2003.

[7] R. Martel et. al., "Carbon nanotube field-effect transistors and logic circuits," in *Proc. DAC*, 2002.

[8] J. R. Heath et. al., "A defect-tolerant computer architecture: Opportunities for nanotechnology," *Science*, vol. 280, pp. 1716–21, June 1998.

[9] S. C. Goldstein and M. Budiu, "Nanofabrics: Spatial computing using molecular electronics," in *Proc. ISCA*, Jul. 2001, pp. 178–191.

[10] N. Cohen et al., "Soft error considerations for deep-submicron CMOS circuit applications," in *IEEE International Electron Devices Meeting:Technical Digest*, 1999, pp. 315–318.

[11] S. Goldstein et al., "Reconfigurable computing and electronic nanotechnology," in *Proc. IEEE ASAP*, 2003.

[12] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, Digital Press, 1992.

[13] J. von Neuman, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies C. E. Shannon and J. McCarthy Eds. Princeton University Press*, pp. 43–98, 1956.

[14] C. N. Hadjicostis, *Coding Approaches to Fault Tolerance in Combinational and Dynamic Systems*, Kluwer Academic Publishers, 2001.

[15] D. A. Spielman, "Highly fault-tolerant parallel computation," in *Proc. 37th FOCS*, 1996, pp. 154–163.

[16] W. B. Culbertson et. al., "Defect tolerance on the Teramac custom computer," in *Proc. FCCM*, 1997, pp. 116–123.

[17] W. Geurts et al., *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*, Kluwer Academic Publishers, 1997.