SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

November 2018, Vol. 61 112102:1–112102:17
https://doi.org/10.1007/s11432-017-9290-4

Fault tolerance on-chip: a reliable computing paradigm using self-test, self-diagnosis, and self-repair (3S) approach

Xiaowei LI[1,2], Guihai YAN[1,2*], Jing YE[1] & Ying WANG[1]

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190*, China;*
[2] *University of Chinese Academy of Sciences, Beijing* 100049*, China*

Received 3 June 2017/Revised 6 September 2017/Accepted 1 November 2017/Published online 24 May 2018

Abstract If your computer crashes, you can revive it by a reboot, an empirical solution that usually turns out to be effective. The rationale behind this solution is that transient faults, either in hardware or software, can be fixed by refreshing the machine state. Such a "silver bullet", however, could be futile in the future because the faults, especially those existing in the hardware such as Integrated Circuit (IC) chips, cannot be eliminated by refreshing. What we need is a more sophisticated mechanism to steer the system back to the right track. The "magic cure" is the Fault Tolerance On-Chip (FTOC) mechanism, which relies on a suite of built-in design-for-reliability logic, including fault detection, fault diagnosis, and error recovery, working in a self-supportive manner. We have exploited the FTOC to build a holistic solution ranging from on-chip fault detection to error recovery mechanisms to address faults caused by chips progressively aging. Besides fault detection, the FTOC paradigm provides attractive benefits, such as facilitating graceful performance degradation, mitigating the impact of verification blind spots, and improving the chip yield.

Keywords fault tolerance, on-chip, self-test, self-diagnosis, self-repair

Citation Li X W, Yan G H, Ye J, et al. Fault tolerance on-chip: a reliable computing paradigm using self-test, self-diagnosis, and self-repair (3S) approach. Sci China Inf Sci, 2018, 61(11): 112102, https://doi.org/10.1007/s11432-017-9290-4

1 Introduction

Reliability is one of the mainstay merits of virtually any computing system. Beyond conventional fault tolerance computing [1], Fault Tolerance On-Chip (FTOC) faces several unique challenges: (1) Resource-limited. FTOC is engaged during the duty time so that any dedicated automatic testing equipment (ATE) are unavailable. Therefore, the only viable strategy is to build-in all required test supports, which makes the FTOC mechanism operate in a self-supporting manner. (2) Overhead-sensitive. Even though todays silicon has become increasingly cheap thanks to the Moores law, it is still unwise to extravagantly use the silicon for non-performance goals. For ordinary users, it is probably highly risky for the chip makers tout for customers with the probability of a system crash rather than the more appreciable performance.

Over the past decade, we have exploited the FTOC to build a holistic solution ranging from on-chip fault detection to error recovery mechanisms [2–6]. We applied them to processing cores, Network-On-Chip (NoC), and on-chip memories. The FTOC framework usually consists of three key components: self-test,

* Corresponding author (email: yan@ict.ac.cn)

self-diagnosis, self-repair, or “3S framework” for short. Some prototypes have been built to demonstrate how FTOC responds to the in-filed silicon degradation. More interestingly, we find that the 3S framework is not only a powerful backbone guiding various FTOC designs and implementations, but also has more far-reaching implications such as maintaining graceful performance degradation, mitigating the impact of verification blind spots, and improving the chip yield. We believe that these design principles will be critical for the chip makers to maintain a competitive edge in the future.

This paper comes from a synthesis of our researches targeting the on-chip fault tolerance in the past decade. We re-visit a series of techniques and propose the “3S” paradigm to govern the diversified techniques which are intrinsically unified for the same purpose: namely hardware fault tolerance. At the edge of Denard scaling [7] and Moore’s law, the IC chips, especially large scale, are prone to suffer more reliability challenges [8, 9]. We believe the proposed FTOC paradigm provides an alternative solution to this challenge. This paper clarifies the key motivation behind FTOC and showcases some key findings, in the hope of attracting more contributions.

The rest of this paper is organized as follows: Section 2 presents the the IC chip lifetime reliability pathology, named as the “Sick Silicon” problem, which is the main application field of the proposed FTOC paradigm. Section 3 presents the limitation of conventional reliability design methodologies and justifies the FTOC paradigm. Section 4 details the major design components of FTOC, followed by the evaluation results in Section 5. Section 6 discusses three far-reaching implications of the FTOC paradigm. Section 7 concludes this paper.

2 “Sick Silicon” problem

The FTOC paradigm can help maintain the lifetime reliability of microprocessors suffering in-field degradation ascribed to both silicon devices and metal wires. The degraded silicon (including the devices and wires) is metaphorically called “Sick Silicon” [4]. According to International Technology Roadmap For Semiconductors (ITRS) projection, silicon aging tops the impending (above 22 nm) reliability challenges. The industry and academic communities have performed significant work to understand the failure mechanisms of semiconductor devices, such as electromigration (EM), bias temperature instability (BTI) including negative and positive BTI (also known as NBTI and PBTI, respectively), time dependent dielectric breakdown (TDDB), hot carrier injection, and temperature cycling, etc. Both NBTI and TDDB draw the most attention concerning several transistor aging mechanisms. Both of them can gradually degrade performance over time. The researchers have evidence that circuit path delay can increase by 10% during the five-year lifetime [10]. Even worse, with technology scaling to the nano-scale, the transistors tend to become more vulnerable and more prone to aging impacts [8].

The integrity of the wires is also degraded. The shrinking size leads to increased current density. Increased current density causes aggravated EM effects, which also contribute to the in-filed performance and reliability degradation. The effects of EM produce increased resistance of the wires and thereby result in increased RC delay. The increasing delay will eventually outgrow the timing margin and, even worse, the wires will eventually breakdown, causing break faults, bridge faults, or stuck-at faults in the chips.

Since the whole chip is exposed to these aging mechanisms, some parts of the chip suffer faster aging than the others. The reason for this is twofold.

2.1 Process variation

The “weak” chips, which are more sensitive to aging, mainly results from process variations [11]. As the feature size relentlessly shrinks generation-by-generation, the impacts of process variations become increasingly evident. Because of the wafer-to-wafer, die-to-die, and within-die variations, the proportion of the circuits with serious mismatches to the golden reference in the design print should be marked as weak silicon and removed from the production batch as yield loss. For example, the typical threshold voltage (Vth) of the transistors may exhibit obvious deviations from the standard settings because of width/length fluctuations caused by an unstable lithographic process. Those transistors with the elevated Vth have smaller tolerances to withstand the aging induced Vth increasing and are therefore prone to be more sensitive to it than those with larger tolerances.

2.2 Unbalanced stressing

The aging rate of silicon devices (including the metal wires) depends not only on the intrinsic constitution of the devices themselves, but also on the stressing duty cycles [12]. From a microscopic perspective, the data patterns, which determine the BTI aging degree, are intrinsically non-uniform across the all bits. Consequently, some transistors are always positively or negatively-biased and therefore degrade much faster than those that are evenly-biased. The heavily-biased BTI aging has a slight chance to enjoy the recovery effect, which exacerbates the aging degree. From a microarchitectural perspective, for another example, the usage of some cores in a multicore processor could be always higher than the others because an oracle round-robin scheduling algorithm is not an option in modern operating system (OS) design. The jobs assigned to different cores can show very distinct stressing degrees. The computationally intensive jobs are usually more power-hungry and therefore generate more heat which can speed up the aging, while those computationally non-intensive jobs are the opposite.

3 Why not screen out weak chips before shipping to the customers?

Since the weak chips tend to age faster than those healthy ones, why not screen them out in the production test stage and only pass those the stronger enough chips. Although this is conceptually simple, it is difficult to implement for two reasons: (1) the blurred boundary between weak and strong, and (2) the prohibitive test cost.

3.1 The blurred boundary between weak and strong

The first reason is pertinent to yield. Qualifying only the strong chips dictates more strict testing standards. Unfortunately, the yield drops sharply, which consumes the profit margin, if we maintain a rather strict testing standard for serious PVT (process, voltage, and temperature) variations. Conventionally, chip manufactures rely on strict chip testing to screen out those unqualified "weak" chips. The procedure relies on the coordination between the designers and the foundry. First, in the design stage, the designers should leave sufficient PVT margin as a safeguard to avoid over-kills. With the statistical process models provided by the foundries, the designers could cover most of the process corners in the design space with a physical level simulation like Monte Carlo analysis. Ideally, SPICE simulators are used to handle process variations from 3-σ to high-σ Monte Carlo simulations. Once the number of local σ is determined by the designer, the worst-case design parameters and the safeguard range are also determined. Then, the foundry could use the built-in-test mechanism or ATE with testing patterns to filter out the bunch of chips that fall outside the safeguard band specified by the designers. In these procedures, the safeguard band or σ setting is a decisive factor determining the yield rate and in turn, the final cost of chip production.

However, as the PVT variation aggravates with process scaling, it becomes less likely to achieve a profitable yield under rather strict parameter specifications. This dilemma can be ascribed to two reasons.

First, the boundary between the qualified chips and the weak chips is becoming increasingly vague. As a result, it is hard to determine a reasonable number of σ as the safeguard band for qualified chips. Taking the speed yield curve in Figure 1 as an example, if one uses the 3σ or 5σ standard to determine the weak chips, they will suffer from enormous yield loss. Otherwise, accepting a less strict safeguard band means more chips with serious fluctuations from the standard reference will be recognized as qualified chips, leaving potentially more underlying threats of in-field failures.

Second, the flattened parameter distribution greatly reduces the population of qualified chips under the previous criteria. For example, Figure 1 displays the trend for two-generation technologies, as reported in [13]. The Gaussian distribution of a parameter specification with a standard deviation σ around the

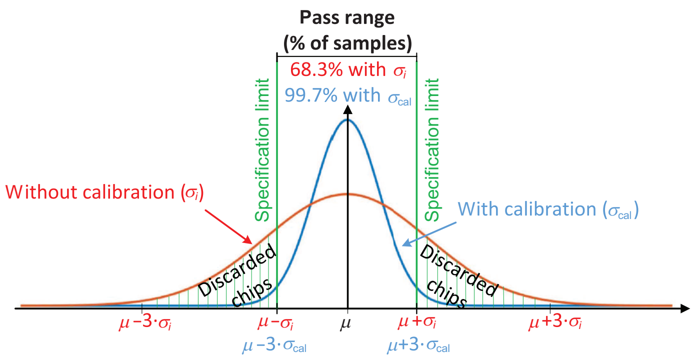

Figure 1 (Color online) Parameter fluctuation distribution with 45 nm and 65 nm process.

mean value μ is shown with the specification limits (3σ in this example). An important observation from Figure 1 is that an increase in variation of advanced technology (red curve) widens the Gaussian distribution, which means a larger proportion of chips that fall within the weak ranges are defined as unqualified. If we use the same standard to filter out unqualified chips, the yield will drop sharply and becomes disastrous for the chip manufacturers.

To summarize, the qualifying boundary increasingly blurs and employing the strict criteria in the hope of maintaining the same lifetime reliability we take for granted probably changes a profitable business into one of futility.

3.2 Prohibitive test cost

Picking out the weak chips does not only challenge the yield, but also challenges the test process. The cost of testing the chips soars at a rapid pace because of the increasing chip scale and complexity. Some new test technologies have been improving overtime. For instance, Xiang et al. [14, 15] proposed to use unicast-based multicast schemes to significantly reduce the cost of core testing in NOCs. However, the trend has slowed down somewhat, but never stopped and even reversed. Dynamic failures and noises, which are much harder to test than traditional static stuck-at faults, cause test overkills due to the accelerated test, and also can create test escapes due to hard-to-cover corners. The trend towards increasing integration and complexity has also been accompanied by technical challenges and rising costs of testing, which can amount up to 40%–50% of the total manufacturing cost. Consequently, using sophisticated techniques such as built-in self tests, design-for-test, and design-for-manufacturability methods, improves the fault coverage from 95% to 99%, but the volume of test patterns, empirically for stuck-at faults, can disproportionately increase by over 50%. The disproportionality is even more prominent for the more harder-to-test transition faults [16]. In addition, testing with extremely high-defect coverage leads to more false rejects and thus lowers the yield.

Besides the test cost issue, accelerated testing sometimes also creates latent weak silicon. The stressful test, also known as burn-in test, is often used to identify the infant mortality as latent defects by stressing the circuit at elevated temperatures and voltages. However, this stressful testing may also cause weak silicon which should have been strong without exposure to such a harsh environment.

In the past decades, one of the critical requirements of manufacturing test is to achieve low test escapes (say, less than 100 defective parts per million (PPM)). However, when circuit variability deteriorates and chip scale grows, it is difficult to use even advanced ATPG to cover all corner cases to locate the subtle manufacturing defects, such as latent timing violation or power delivery noises that are difficult to trigger in testing. If we utilize a strict test escape rate, say 100 PPM, we can expect the test cost per chip to likely increase dramatically to achieve the last several percentages of fault coverage due to an obvious long tail effect [17].

Therefore, it is unavoidable that the chip products will be accompanied by the weak chips, or more specifically, the chips with weak parts. Hence, these hard-to-catch culprits will greatly threaten the reputation of the chip providers. "We have to take action before it is too late", and the action is to equip the chips with a self-supportive FTOC mechanism.

4 FTOC: a 3S approach

As a fault tolerance mechanism, FTOC has the ingredients of generic fault tolerance mechanisms: fault detection, fault diagnosis, and error recovery. Fault detection is used to judge whether the system suffers from erroneous executions, then fault diagnosis digs deeper to determine where and how such errors happen, which is followed by a recovery routine to correct the error to the expected outcomes. For the FTOC, the generic framework evolves with several new attributes which provide the essences of the self-supportive 3S approach.

4.1 Self-test: low-cost built-in fault detection

The fault detection, which is virtually realized with dual-module redundancy either in spatial or temporal dimensions, is not viable due to its notoriously high overhead in terms of hardware or performance. For example, there are many fault detection schemes based on thread-level redundancy (TLR), core-level redundancy (CLR), and execution-level redundancy (ELR). Both TLR and CLR detect faults at the expense of computing throughput, a typical spatial dimension overhead. Furthermore, ELR virtually needs re-execution of the code and thereby dictates a large temporal overhead, even though such strict redundancy schemes promise perfect detection coverage.

To enable FTOC, we must resort to more thrift detection approaches. To achieve this, what we can compromise is the perfect detection coverage, given that the principal objective of FTOC is to isolate the Sick Silicon, rather than protect every instruction from fault contamination at all times. We design a highly cost-efficient self-test with respect to a probabilistic principle, rather than a deterministic principle. The detection routine should not take a significant number of processor cycles, and should be as transparent as possible to the kernel and user threads. Symptom-based fault detection which is built upon low-level circuit timing monitoring can fulfill this purpose [5, 18].

In symptom-based fault prediction, a symptom is defined as a signal stability violation. Basically, the stability violation of a signal is defined as at least one transition happens in the time interval during which the signal should be kept stable. A setup time violation, ascribed to progressive silicon aging for example, is a type of typical stability violation. As Figure 2(c) shows, in a clock cycle, we should reserve a timing span, that is a safeguard band, to meet the minimal setup time requirement. For the degradation-free case, there should be no single transition during the safeguard band; by contrast, if the transistors involved on the relevant timing paths suffer sufficient aging, the transition-free condition can no longer hold. By detecting the transitions in the safeguard band, the impending faults can be detected. Of course, whether an aged circuit can result in a stability violation is determined not only by the “sickness” of the silicon, but also by the data patterns which can sensitize the corresponding timing paths. However, the timing paths of Sick Silicon will show a much higher probability than healthy silicon to trigger the stability violations. By detecting the distribution of the stability violation, we can discriminate the sick parts from the healthy parts.

The key instruments to detect the stability violation is timing sensors, which are commonly based on dynamic circuits satisfying sub-nanosecond to even tens of picoseconds detection resolution. Figure 2 shows a sensor design. The basic stability checker (Figure 2(a)) can be derived from a sensing circuit for on-line delay fault detection, in which the integrity of the signal (S) is verified by a pair of exclusive nodes ($S1$ and $S2$), a stability violation will discharge the charged node and thereby cause both nodes to be at the “0” state, which signifies a timing violation. The outputs are compacted with a dynamic NOR for reducing the number of output latches (Figure 2(b)), where the $M11$ and $M12$ serve as a level restorer for node X.

Multiple timing sensors are embedded in the host chip during fabrication. These sensors collectively form a monitoring system with fine-grained spatial detection resolution. The problematic component, such as an arithmetic logic unit (ALU), or a L1 cache bank, can be pinpointed. These faulty components

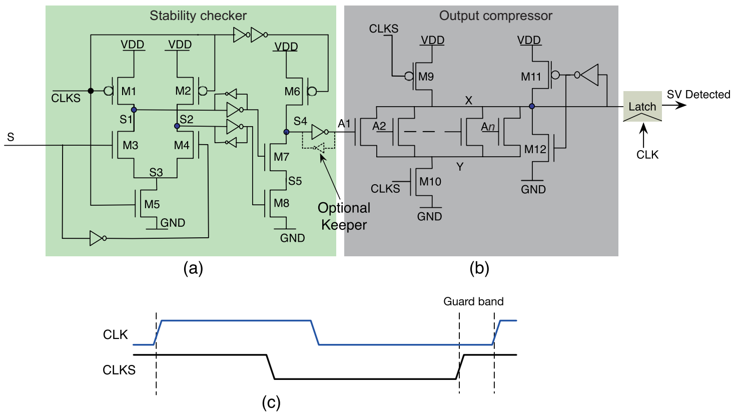

Figure 2 (Color online) Timing sensor design. (a) Stability checker; (b) output compressor; (c) clock timing.

can be masked from the other healthy parts, simply like a patient undergoes a surgery. These circuit-level adaptations can be automatically executed transparently on the host operation systems.

4.2 Self-diagnosis: accurately pinpointing in situ faulty components

In FTOC, the diagnosis has two objectives: (1) pinpointing which components have been suffered permanent faults, and (2) estimating the level of performance degradation will be taxed due to the faults. Before delving into the details, we would like to first clarify the key differences between diagnosis in FTOC and conventional chip diagnosis routines.

The diagnosis routine in FTOC, called self-diagnosis, is very different from conventional diagnosis used in the yield learning phase, in terms of objective, techniques used, target granularity, and fault models.

- First, self-diagnosis is used to identify and locate the malfunctioning components, while the diagnosis in the production phase is mainly to help locate the defective physical or electrical contexts [19]. The designers refine the physical designs to avoid these cases to ramp up the yield learning rate.
- Second, the self-diagnosis intrinsically relies on built-in logic to locate the defective component, while the conventional diagnosis heavily relies on the silicon scan test and is conducted off-line by using sophisticated logical diagnosis tools.
- Third, the granularity of self-diagnosis uses relatively coarse-grained components, such as core-level granularity, which have independent functionality and are usually loosely coupled with other parts, while the conventional diagnosis works at much finer-grained granularity at the logic gates or standard cells. In other words, self-diagnosis is based on functional testing and conventional diagnosis is based on structural testing.
- Accordingly, the fault models of self-diagnosis describe the malfunction of components and therefore are more ad hoc, such as parity mismatch in the ALU components, while that of conventional diagnosis targets more silicon-level imperfections, such as bridge, open, abnormal leakage.

For FTOC, determining which parts of a chip got sick usually is trivial with the fine-grained self-detection facility. If the corresponding timing sensors keep alerting stability violations, the faulty components can be switched off to avoid erroneous computations. In this case, the diagnosis and associated repairs are trivial. From Figure 3, for example, there are four homogeneous ALUs in the processor core, the diagnosis agent logs the number of alarms reported by the self-test procedure. Each logging period can be as long as days or weeks to improve the diagnosis confidence level. By analyzing the alarm distribution, the self-diagnosis agent can discriminate the faulty ALU. In this example, the alarm density ascribed to ALU2 is significantly higher than the others, so the diagnosis agent marks that ALU2 should no longer be available anymore. The computation is thereby offloaded to the remaining three health ALUs. Consequently, this core will continue to work at the degraded performance level. The similar diagnosis logic can be also applied in the core-level, especially for many-core processors.

It is more challenging to determine the performance impact given the faults detected, because the

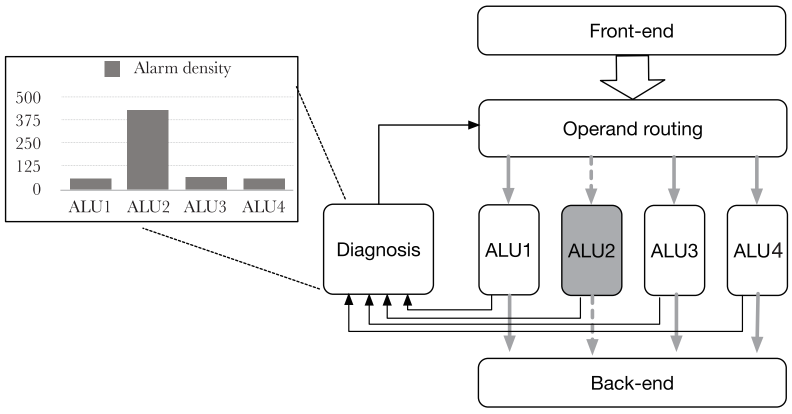

Figure 3 Exemplify self-diagnosis logic for a 4-ALU processor.

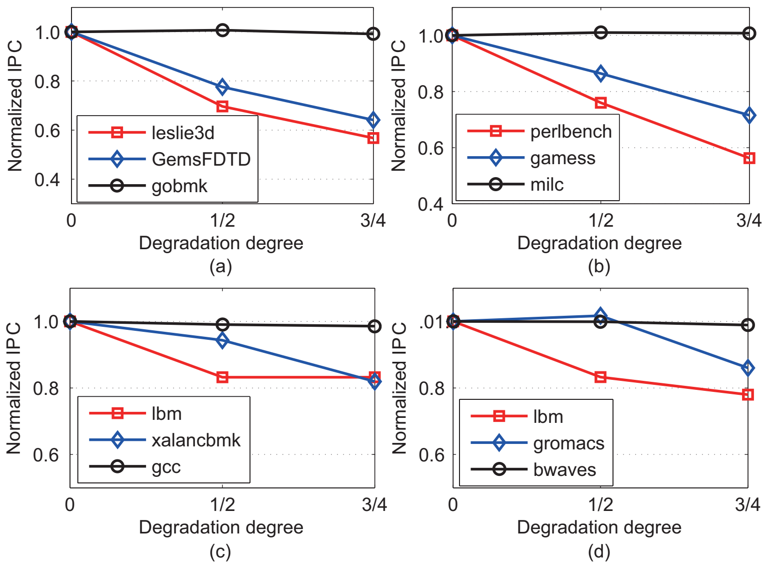

Figure 4 (Color online) Performance degradation vs. defect degrees of (a) instruction window, (b) L1 instruction cache, (c) L1 data cache, (d) L2 cache, where the following SPEC CPU2006 benchmarks are used: leslie3d, GemsFDTD, gobmk, perlbench, gamess, milc, lbm, xalancbmk, gcc, gromacs, and bwaves.

performance degradation depends on both the applications and the extent of the defects. For example, Figure 4 shows the performance responses of the cores under various types of degradation. The cores are salvaged from instruction window defects, or L1 instruction/data cache defects, or L2 cache defects, respectively [20]. For simplicity, we do not show the more complicated compound defects. The degradation degree of "0" indicates defect-free, and 1/2 indicates a half resource is unavailable, and so on. The results show that the performance response not only depends on the degree of degradation, but also exhibits to be highly application-specific. For example, the `gobmk` (a SPEC CPU2006 benchmark) in Figure 4(a) shows to be very resilient to the instruction window degradation; however, by contrast, the `leslie3d` and `GemsFDTD` are very sensitive to it. Such complexity is never unique for the instruction window only, but also to other resources, as exemplified in the other three sub-figures. Hence, even though the defect and associated defect level are accessible to the OS, we still have no ways to determine the level of performance impact such a degradation causes the running applications.

Yan et al. [4] proposed the CoreRank approach to address this challenge. The CoreRank quantifies the core-level performance degradation towards more meta-program representations, called snippets, which are dynamic micro-operation streams and are oblivious to all the software level interferences. The snippet can be readily characterized by built-in performance counters, without any instrumentation into the running workloads. The performance of core C_i on the snippet S_m is denoted as $P(C_i|S_m)$, which can be

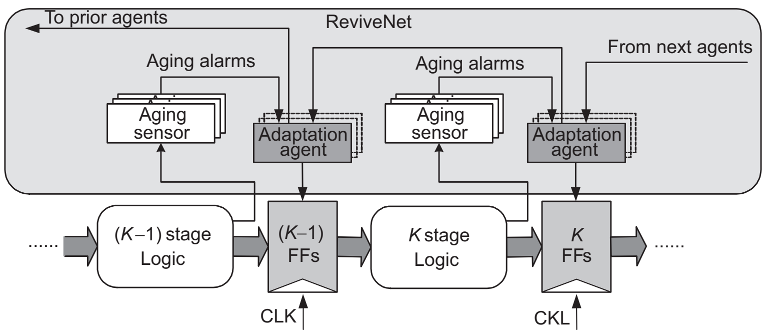

Figure 5 Circuit-level rejuvenation with timing adaptation.

obtained by reading the corresponding performance counters [21]. If $P(C_j|S_m)$ differs from $P(C_i|S_m)$, the relative degradation can be easily extrapolated as the ratio of $P(C_j|S_m)$ to $P(C_i|S_m)$. Given any running program is composed of a sequence of various meta-programs (snippets), the program-level performance degradation can be estimated by aggregating the degradation on each individual snippet. Please refer to [4] for more details.

For FTOC, the diagnosis is triggered only when the test procedure prompts the alarms. To minimize the overhead, one diagnosis agent can be shared by multiple timing sensors in a round-robin manner [2] controlled by a finite state machine. To minimize the penalty of power and performance in the fault-free scenarios, the diagnosis procedure is not always on, but is periodically invoked by abnormal states such as a machine crash.

4.3 Self-repair: rejuvenating the degraded microprocessors

Generally there are two types of core-salvaging approaches: (1) Fault isolation. Decoupling the faulty components [20] can avoid execution contamination and maintain a graceful degradation of performance. (2) Adaptive voltage-frequency setting and timing recycling [22]. For example, if the critical path delay increases due to aging, the functionality is maintained provided the working frequency is slowed down to accommodate the extra delay. The self-repair can be implemented at three abstract levels: circuit level, microarchitectural level, and architectural level. But we should note that such a classifying scheme is never strict, but only provides a roughly categorical image for easier understanding.

4.3.1 *Rejuvenation at the circuit level*

Figure 5 illustrates a circuit-level pipeline with a rejuvenation facility. Each stage is monitored by a set of periodically-invoked aging sensors used to detect the signal transitions in the safeguard band. The aging sensors are deployed to monitor the critical paths. In the fault-free scenarios, no transitions could happen in the safeguard band, but after suffering from aging, some transitions could be delayed into the safeguard band, represented as a stability violation [5], a type of faulty symptom. With the awareness of aging, we can accommodate the impending aging failures by adapting localized timings.

The adaptation to each stage is implemented with a set of time-borrowing agents which are fed by not only the local stages aging sensors but also the next stages agents, thereby enabling bidirectional adaptation, namely backward timing adaptation (BTA) and forward timing adaptation (FTA). The BTA uses the $(K + 1)$st stages timing slack to accommodate the aging emergencies in the Kth stage, while the FTA uses the $(K1)$st stages slack to accommodate the emergencies in the Kth stage. When an aging sensor detects an alarm, the BTA, FTA, or BTA and FTA can be simultaneously enabled to tolerate this aging delay.

4.3.2 *Rejuvenation at the microarchitectural level*

The microarchitectural rejuvenation largely relies on decoupling the faulty components from the remaining healthy parts, or reconfiguring the microarchitectures [20]. The components which can be readily modified to be reconfigurable include the ALU arrays, cache banks, and register files. They share the

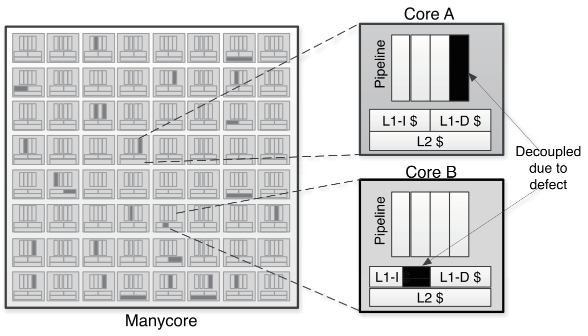

Figure 6 Microarchitectural rejuvenation.

common feature of regularity with intrinsic spares. The repaired procedure is also similar: marking the faulty component as unavailable so it will never be allocated to dynamic instructions. With some more sophisticated circuit techniques, these components can even be totally decoupled form the power grid, thereby preventing them from leakage.

For example, as shown in Figure 6, the Core A and Core B suffered a pipeline defect and an L1 I-cache defect, respectively. The defect-affected partitions, marked as dark parts, are decoupled from the rest to make each core functionally correct, but in a degraded manner.

Such microarchitectural approaches are also common in on-chip memory subsystems, caches or scratchpad memory for example. We can use the bank remapping method to cure the coarse-grained non-uniform cache access (NUCA) cache failure within the framework of self-test, self-diagnosis, and self-fault-isolation. It utilizes the routing logic in the Network-on-Chip to transparently remap the physical space associated with the failed banks to healthy cache banks, so that the system will not see the cache failure and maintain a wholesome physical memory space on-chip.

In fact, such microarchitectural approaches are more common in on-chip memory subsystems. A cache or scratchpad memory, always occupies a significant proportion of silicon real-estate. Using the last level cache for example as the failure mechanism in the SRAM cells, it suffers from different fault models, such as permanent read/write failures due to the SNR issue, retention fault or single-event upset (SEU). Regarding the granularity of the cache failures, it includes conventional bit failures that could be cured by error correction bit or remapping-based bit repairing, and array or bank failures that occur in large-scale cache structures like distributed NUCA architectures. Fine-grained cache failures can be cured with conventional error correction or bit/row/column replacement. However, in modern large scale chip multi-processors, bank-level failures due to interconnection issue or isolation requirements are less discussed. For example, when a NoC-node is isolated from a resilient chip multiprocessor, it also creates inaccessible NUCA cache banks because of the connectivity issue, which should be tolerated to enable a degradable cache system. We propose a bank remapping method to cure the coarse-grained NUCA cache failure within the framework of self-test, self-diagnosis, self-fault-isolation. It utilizes the routing logics in Network-on-Chip to transparently remapping the physical space associated to failed banks to healthy cache banks, so that the system will not see the cache failure and maintain a wholesome physical memory space on-chip. Furthermore, to reduce the negative impacts imposed by the bank failures, our work uses a utility-driven remapping policy to match the failed cache banks to an under-utilized cache bank, so that the system receives the least performance penalties caused by the bank failures. The remapping method relies on a dynamic stack-distance analyzer to measure the space utility of different address spaces and keeps on remapping the failed banks to their favored compatible healthy banks. In this way, the bank-sharing induced block conflict will be reduced and the conflict-induced eviction cache miss rate will be minimized. The whole framework guarantees that the bank failure will be tolerated with a very small performance cost. When future systems are built with unstable devices or an unstable environment, such an inexpensive fault tolerant mechanism is very useful.

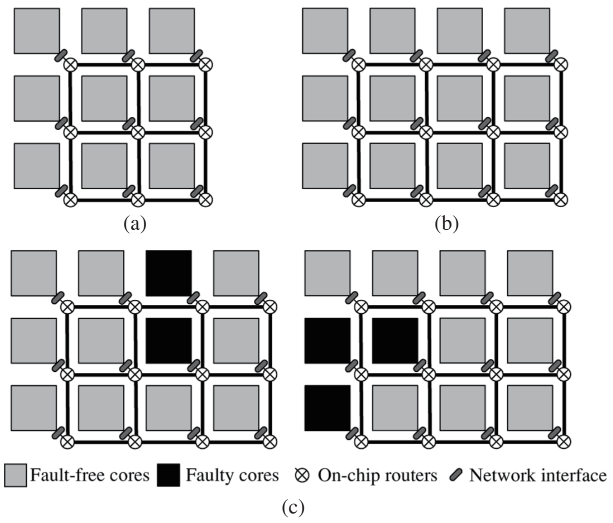

Figure 7 Topology reconfiguration-based architectural level rejuvenation for a manycore. (a) The topology demand; (b) the topology with spare cores; (c) the topology with faulty cores.

4.3.3 *Rejuvenation at the architectural level*

The architectural rejuvenation is usually conducted at the core-level. There are two major rejuvenation styles: topology-invariant and topology-reconfigurable approaches, where the topology refers to the NoC topology connection of tens even hundreds of cores.

Using core level DVFS to tolerate a cores degradation is a typical topology-invariant approach. The cores initially have the same maximum frequency, Fmax, but with the in-field aging degradation, the Fmax of the cores can differ from each other. If a core ages with a prolonged critical path delay, we can scale down the cores frequency to maintain safe timing, at the expense of more sophisticated per-core DVFS. Meanwhile, the topology, that is the cores location related to other cores, remains intact. The topology-invariance can simplify the NoC implementation and traffic management.

However, if a core suffers an unrepairable failure, we must either map it out of the healthy region, or find a substitute. In either case, the topology must be changed and topology-reconfigurable approaches must be employed [3, 6]. One typical solution is called $N + M$ paradigm, i.e., there are N normal cores, which are visible to the OS, and M spare cores, which only serve as substitutes for failed cores and are invisible to OS. The similar solution is adopted in the "Cell" processor (N=7, M=1), where an N-core processor is provided with M redundant cores and we always provide customers with N operational cores. The spare cores are viewed as overhead. However, as the number of on-chip cores increases, the overhead of leaving a few redundant cores on-chip unused is acceptable because a single core is inexpensive compared to the entire chip.

In fact, the industry has started to employ core-level redundancy in their products. Even though the objective is mainly for yield or performance, a similar rationale should be also applied to enhance the lifetime reliability. In such a case, rejuvenation is about substituting the faulty cores with the spares. The topology determines the ideal performance whereas the routing algorithm and the flow control mechanism determine how much of this potential is realized. However, when the failure cores are replaced by spare cores, the topology of the target design can be different. For example, suppose we want to provide 9-core processors with a 3×3 2D-mesh topology, as shown in Figure 7(a). Also, suppose three redundant cores (1 column) are provided, as shown in Figure 7(b). If some cores (no more than three) are defective, we could still get 9-core processors.

However, from Figure 7(c), if the faulty cores are replaced by the spare cores, not only are the topologies different from what we expect, but also the topologies of different chips can be very distinct. Consequently, there is a mismatch between the logical topology, 2D-mesh in this example, and the physical topology, namely the topology with the disabled cores. Clearly, there could be many ways to map the logical

Table 1 Degradation models

	Degradation component	Decoupled capacity
Front end	Branch predictor	1/4 : 1/2 : 3/4
	instruction window	1/4 : 1/2 : 3/4
Back end	Issue width	1/4 : 1/2 : 3/4
Memory	L1 data cache	1/4 : 1/2 : 3/4
	L1 instruction cache	1/4 : 1/2 : 3/4
	L1 D-Cache	1/4 : 1/2 : 3/4
	L2 cache	1/4 : 1/2 : 3/4

Table 2 Core configuration

Parameter	Value
Frequency	1 GHz
L1 I/D cache 32 KB	Cache line 64 B, associativity 4
L2 cache 512 KB	Cache line 64 B, associativity 8
Issue width	4
Branch predict entry	1024
Instruction window	96

topology to the physical topology. So, the challenge in the $N + M$ paradigm is to determine which topology is optimal. The problem has been proven to be NP-complete and can only be solved with a heuristic algorithm [6].

5 Evaluation

Since the FTOC paradigm is mainly employed to handle in-field chip aging faults, we first build a degradation model to capture the implication of aging to chip performance. Then, we use a manycore processor as the baseline to study how FTOC responds to in-field faults in terms of performance and overhead.

5.1 Experimental setup

First, we list the degradation models used in the experiments in Table 1. The degradation terminology is borrowed from [23]. Basically, the degradation is roughly divided into three categories: (1) front-end degradation, involving a branch predictor and an instruction window; (2) back-end degradation, reflected by throttling the issue width; and (3) on-chip memory degradation, involving private L1 and private L2 caches. For each component, we assume three degradation degrees: mild, medium, and severe, corresponding to 1/4, 1/2, and 3/4 capacity disabled (in this way to mimic the impact of aging-affected micro-architectural components), respectively. The degradation models exclude the extreme cases of 0 and 1, corresponding to defect-free and totally unresponsive components that have no possibility to be salvaged.

The system performance is evaluated with Sniper [24], a multi-core simulator based on the interval core model [17] and Graphite [25] simulation infrastructure. Sniper can accurately simulate x86 architecture at a speed of millions of instructions per second. Sniper uses the Intel Pin tool (version 61147) to dynamically profile the stream of uops of each core, which provides an easy way to collect instruction snippets in the experiments. We use a 12×12 manycore as the baseline to evaluate the performance. The core configuration is shown in Table 2. The performance is measured by summing up all of the active core IPC within the same time window. When measuring the performance degradation, we use the geometric mean of slowdown of all cores to implicitly consider fairness. For the processor without FTOC support, it is reasonable to assume that the applications are randomly mapped to the cores with different degradations, since we have no way to distinguish which core is superior to another. By contrast, for those with FTOC, the minimal degradation can be achieved for a given workload, thanks to the performance self-diagnosis facility. In addition, when comparing the performance, we neglect the scheduling overhead because the execution time of a snippet is very comparable to a Linux OS time slice in reality.

5.2 Workloads

We build two types of workloads: SPEC CPU2006 for the multi-program workloads, and PARSEC for the multi-thread workloads. The manycore is always fully loaded with the synthesized workloads. For example, for the multi-program workload, we randomly chose 12×12 benchmarks from the SPEC benchmark suite to feed the manycore. For the multi-thread workloads, we use 4-thread and 8-thread configurations, respectively. Note that a specific benchmark can be repetitively chosen from the benchmark suites. To create diversity, each benchmark is fast-forwarded to different phases before sampling the snippets.

5.3 Performance degradation

We study the performance degradation while oblivious to the core healthy conditions. Figure 8 shows the normalized performance degradation under gradually increasing percentage of unhealthy cores. The degradation model for each core is randomly chosen from the degradation models listed in Table 1, with "mild", "medium", and "severe" degradation degrees, respectively. We run 500 workloads to highlight the statistical trend, presented by the violin plot[1]. We can draw the following conclusion from the experimental results.

• The result confirms the unhealthy cores can dramatically degrade the system performance. Specifically, the multi-thread workloads are more sensitive than the multi-program workloads to core degradations, because of the more prominent "cask-effect" in the multi-thread workloads. The more thread-level parallelism is the more sensitive to core degradation. Also, we find the performance degradation progresses relatively slowly, especially when the population of degraded cores exceeds 60%. This is because every multi-thread workload has a high possibility to be slowed down by at least one unhealthy core. Given the prevalent big-data processing today dominated by multi-threaded programming models (MapReduce for example) and algorithms, the impact of unhealthy cores should not be underestimated.

• FTOC can help hide performance degradations, as Figure 9 shows. The scheduling policy directly relies on FTOC self-diagnosis results to guide application mapping. The results show that even the population of unhealthy cores is 50%, the performance declines no more than 10% , compared to around 55% degradation without FTOC support. We also conclude that it is impractical to expect CoreRank to help revive a "terribly sick" processor, i.e., majority cores are salvaged. From Figure 9, when the unhealthy core population goes beyond 50%, the performance degrades quickly. The performance benefit is very slight when the population of unhealthy cores surpasses 70%. Nevertheless, below 50%, the processor, even with unhealthy cores, can provide performance very close to a fresh one.

• Impacts of degradation degrees. Figure 8 shows the results of performance under mild, medium, and severe degradation, respectively. Unexpectedly, we find the performance under mild and medium degradation is very similar: the difference is merely 3%–5% at each unhealthy population. This is because most of the applications cannot fully exercise even the medium cores, so the mild and medium cores make a slight difference in performance. But the severe cores cause more appreciable performance degradation. An interesting difference emerges when we engage FTOC. Figure 9 shows that even though the performance decreases with an escalated number of faulty cores, the takeoff points of degradation are different from each other. The more severer the degradation, the earlier is the appearance of the takeoff point. For example, for mild degradation, the performance will not decrease until around 50% unhealthy cores is attained. While for severe degradation, 30%–40% unhealthy cores can cause performance degradation.

1) A violin plot is a statistical illustration of a group of values; the density of each value is reflect by the width on a corresponding notch.

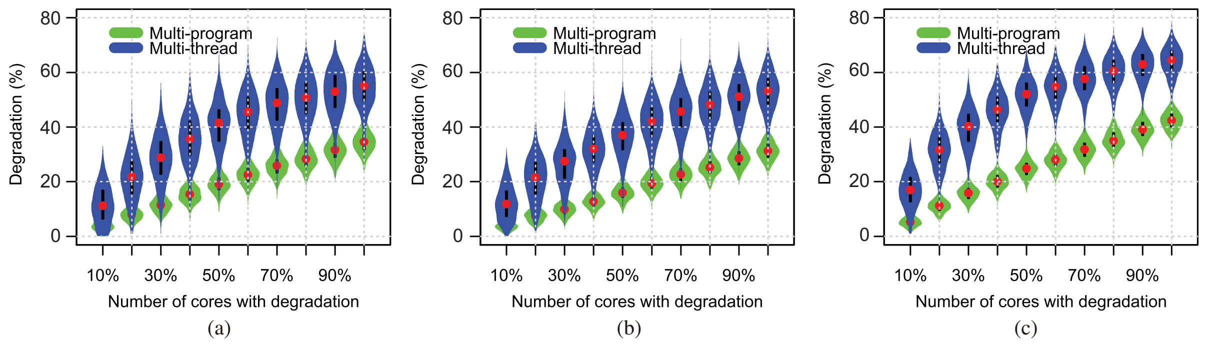

Figure 8 (Color online) Performance degradation for the processors with (a) mild (L), (b) medium (M), and (c) severe (R) degradation models, without FTOC, 4-thread for multi-thread workloads.

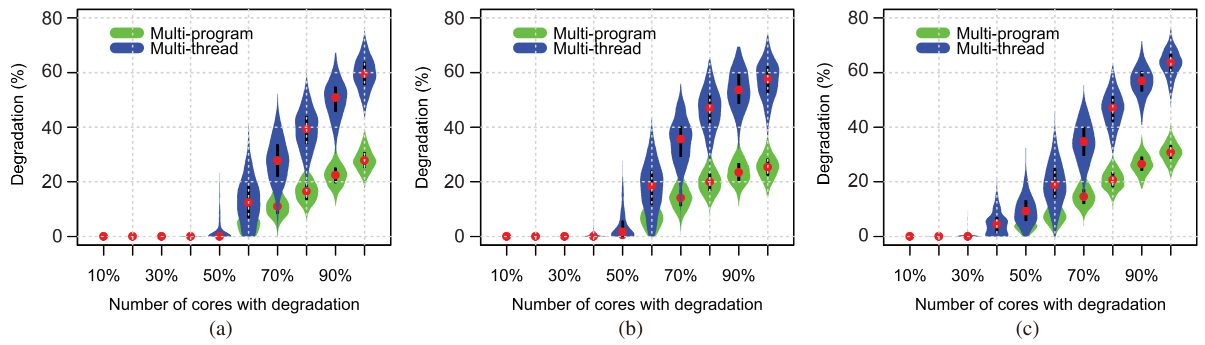

Figure 9 (Color online) Performance degradation for the processors with (a) mild (L), (b) medium (M), and (c) severe (R) degradation models, with FTOC, 4-thread for multi-thread workloads.

5.4 FTOC-enhanced reliability and maintainability

The above experimental results have shown that FTOC provides graceful degradation under defective cores. The reliability is therefore enhanced by smartly circumventing the defective components that have either intrinsically redundant spares or intentionally reserved spares [6]. The salvaging routine is governed by FTOC and is transparent to the user-visible software. At the same time, the performance degradation of a FTOC-equipped system serves as a key indicator for maintenance. If the performance loss cannot be recovered by the self-repairing routine, the operator should take control by initiating a replacement when the degradation exceeds some performance acceptance threshold. The operator can take time to do so given the process of degradation is immune to any user-visible errors and time non-critical.

5.5 Hardware overhead analysis

Because of the intrinsic built-in nature of FTOC, the hardware overhead is unavoidable. However, given that an absolute overhead percentage provides little meaning because it highly depends on the host design with which FTOC affiliates. In the following discussion, alternatively, we will elaborate on the amount of overhead a single functional unit will dictate, with which the system level overhead can be easily estimated given the specific host designs. In our prototype design, the overhead mainly originates from the self-test and self-diagnosis, described as follows.

To support the self-test functionality, we have built a set of timing sensors. Each sensor consumes about 36 transistors, almost equivalent area dictated by a full-fledged modern register design. These sensors will be organized as a sensor tree with logic OR gates, and are still very area-efficient.

To support the self-diagnosis functionality is more costlier, mainly in terms of supporting the performance-diagnosis facility. Since each snippet's signature (usually encoded with several bytes) and associated performance value (measured using instruction per cycle (IPC)) must be registered, we use a hash-table to fulfill this purpose. The main hardware overhead comes from the bloom filters and MD5

Table 3 Self-diagnosis hardware overhead

Component	Combinational logic gates (K)	Storage (KB)
MD5	76	0.1
Bloom filter	1582	12.6
Pearson Hash	0.62	0.6
Bucket	0.38	7.3
Others	61.2	0.2
Total	1720	20.8

hash function. The study shows that a 512-bucket hash table is large enough. The detailed overhead includes hash functions, bloom filters, MD5 registers, and some associative search logics. The key logic is implemented into RTL with Verilog, and synthesized it with the Altera Quartus tool. The overhead is shown in Table 3. The results show that a FTOC imposes about 1.7 M logic gates and 21 KB storage. This overhead is small compared to a processor with billion transistors.

6 Discussion: three far-reaching implications of FTOC

The FTOC paradigm has great potential to critically complement the state-of-the-art IC design. However, we should note that the specific techniques mentioned above should not be supposed to be comprehensive, but the concept of the 3S-based FTOC framework can be tailored for more purposes. We summarize three perspectives in the following section.

6.1 Maintaining graceful degradation

Faults could happen during the lifetime of a system. If the faults are transient, the system may be recovered by rebooting. However, if the faults are permanent, some resources of the system, such as cores in multi-core processor or interconnections of NOC will no longer be functionally correct. Without isolating the faults, the whole system may even turn off completely. However, by detecting, diagnosing, and isolating the faulty components, the system may still be able to work correctly using the remaining good components, though at a lower performance degree, i.e., Graceful Degradation. No redundant components are assumed, which means the components of the system have already satisfied the capability of reconfiguration for correct function. It is not surprising that these two design philosophies converge since they share the same objective. There are two key questions required to be answered for the FTOC-based graceful degradation: (1) what is the granularity? (2) how to implement it? The FTOC mechanism sheds light on the answers.

The processor core and the NoC interconnection are two typical reconfigurable components used in graceful degradation. In multi-core processors, when one core is faulty, other cores can still function. In the NoC, when one interconnection node is faulty, other nodes may substitute its routing function. With FTOC, there are more redundant resources to keep the whole system working properly. More fine-grained components can also be considered. For example, a redundant arithmetic logic unit (ALU) can be added to a core, so when one ALU fails, the core can still work correctly.

Using more fine-grained components for fault tolerance and performance degradation can improve the lifetime of the system, but its disadvantage is the hardware cost, not only including the hardware for isolating the faulty components, but also including the hardware of detecting such fine-grained components. However, FPGA is an exception, since it is programmable. Hence detecting the faulty Look-Up tables (LUTs), interconnection boxes, or other fine-grained components of FPGA can be realized by specific circuits, and isolating the faulty resources can be achieved using placement and routing constraints while designing FPGA circuits. Hence, it is possible for FPGA to perform fine-grained analysis without any hardware overhead, but with a performance penalty.

Furthermore, FTOC provides more possibilities and opportunities for effective graceful degradation. The implementation of graceful degradation requires accurate diagnosis of the faulty components. This is the same in FTOC. For example, in NOC, it is necessary to diagnose the switch, the router, the link, and so on [26]. With the knowledge of locating the faulty components, the routing for graceful degradation is an optimization process with the constraint that the faulty interconnections should not be used. With more faulty components, more constraints exist in the optimization problem, so its solution, i.e., the performance, will become progressively worse, until it reaches a limit that no available routing can be found, and then the whole system will fail. Moreover, FTOC can make the graceful degradation more simplified and effective. If an interconnection node has two routers, one of which is redundant, then when the working router fails, the node can simply switch to the redundant one. In this case, the routing delay remains similar, so the performance is maintained.

6.2 Helping fix some verification blind spots

Modern designs have become more complicated, which poses serious challenges for verification. The verification techniques cannot scale to the complexity of the modern designs, so some bugs could escape from verification and remain in the silicon. If the bugs really exist, they are like the permanent faults. If these faults are not detected during testing, the products with bugs will enter the market. If the bugs are encountered by customers, it will be a large financial loss to recall the chips. In this situation, FTOC is an alternative method to fix the problem. FTOC has at least two benefits for verification: (1) locate the verification blind spots; and (2) fix the escaped bugs.

From the perspective of behavior, the escaped bugs are like the permanent faults. Both cause the system to work incorrectly. In FTOC, the preliminary step for isolating faults is to detect and diagnose the faults. The same function is suitable for finding the escaped bugs. Learning how and why the bugs escaped from the adopted verification techniques is important for improving the verification process and avoiding the similar bugs remaining in silicon. A more fine-grained fault detection can provide more precise information about the escaped bugs. For example, it is easy for the designers to learn the escaped bugs by informing them just the ALU is faulty than informing them the whole processor core is faulty. Therefore, the detection circuit for escaped bugs should be designed properly. For example, the detection circuits can be inserted in some critical points in the control flow [27].

In FTOC, the faults are isolated to allow the whole system to work correctly. Some bugs can also be isolated. However, since the bugs may be repeated, isolating the bugs may not be effective. For example, in the multi-core processor, if all the cores have the same design, they will contain the same bugs as well, so it is meaningless to isolate faulty cores. Under this scenario, there are three ways to correct the bugs. First, heterogeneous cores can be designed so that even if one type of core contains bugs, other types of cores may still function correctly by isolating the faulty cores. But this method may result in a large performance losses, since a portion of the cores are unavailable. Second, during the design, combined with fine-grained bug detection circuits, some configurable components can be inserted into the critical locations [28]. If bugs are detected, specific configuration bits can be downloaded to the configurable components to correct the circuit behavior. Furthermore, in the CPU+FPGA SoC, if some bugs exist in the computing components, it is also possible to use the FPGA as a fault-tolerant component to perform the correct function. Third, for some bugs, it is also possible to use software-hardware cooperation to bypass the bugs [29]. For example, if there is a bug in the subtraction computation hardware component, the OS can compile a subtraction operation into an add operation. In this way, the same function is performed by detouring the bugs. Therefore, using the above methods, with properly inserted bug detection and recovery design, some escaped bugs from the verification phase can still be fixed after the chips are manufactured and sold to the customers, though with some performance degradation.

6.3 Improving gross yield

Bugs may escape from verification, and defects may happen during manufacturing. There are mainly two types of defects: the permanent defect and the transient defect. The permanent defects such as stuck-at faults will permanently affect the behavior of the chip. More specifically, they destroy the Boolean relation within the chip. In certain situations, the chip or the corresponding component will definitely fail. The transient defects, such as small delay defects, are types of timing faults. The chip only fails under a certain condition. Different from bugs which may be repeated, e.g., in multi-core processors, the cores with the same design have the same bugs, the defects do not have such characteristics. Hence the FTOC can also tolerate some defects.

The chips with defects are considered as faulty chips, but if the defects can be tolerated, then the chips can still work correctly and be considered as good chips. Hence, the yield can be improved, but the promised performance may be degraded.

7 Conclusion

The FTOC is an incorporative framework to build synergy among many advanced fault tolerance-oriented techniques. In this paper, we placed self-test, self-diagnosis, and self-repair (self-recovery) into a unified framework, namely the “3S” framework, and clarified the difference from their conventional counterparts whenever possible. We use the manycore undergoing various degrees of aging faults as the baseline to show the efficacy of FTOC, and discuss three far-reaching implications in terms of graceful degradation, verification, and yield.

Although we have made some initial attempts to solidify this framework, the potential has never been fully exploited. We believe FTOC can help deliver more reliable SoC systems suffering in-field degradation in the future.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. 61532017, 61572470, 61521092, 61522406, 61432017, 61376043), and in part by Youth Innovation Promotion Association, CAS (Grant No. Y404441000).

References

1 Borkar S. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro, 2005, 25: 10–16
2 Yang G H, Han Y H, Li X W. ReviveNet: a self-adaptive architecture for improving lifetime reliability via localized timing adaptation. IEEE Trans Comput, 2011, 60: 1219–1232
3 Fu B, Han Y, Ma J, et al. An abacus turn model for time/space-efficient reconfigurable routing. In: Proceedings of the 38th Annual International Symposium on Computer Architecture, San Jose, 2011. 259–270
4 Yan G, Sun F, Li H, et al. CoreRank: redeeming “Sick Silicon” by dynamically quantifying core-level healthy condition. IEEE Trans Comput, 2016, 65: 716–729
5 Yan G, Han Y, Li X. SVFD: a versatile online fault detection scheme via checking of stability violation. IEEE Trans VLSI Syst, 2011, 19: 1627–1640
6 Zhang L, Han Y H, Xu Q, et al. On topology reconfiguration for defect-tolerant NoC-based homogeneous manycore systems. IEEE Trans VLSI Syst, 2009, 17: 1173–1186
7 Dennard R H, Gaensslen F H, Rideout V L, et al. Design of ion-implanted MOSFET’s with very small physical dimensions. IEEE J Solid-State Circuits, 1974, 9: 256–268
8 Srinivasan J, Adve S, Bose P, et al. The impact of technology scaling on lifetime reliability. In: Proceedings of International Conference on Dependable Systems and Networks, Florence, 2004. 177–186
9 Borkar S, Karnik T, Narendra S, et al. Parameter variations and impact on circuits and microarchitecture. In: Proceedings of Design Automation Conference, Anaheim, 2003. 338–342
10 Wang W P, Yang S Q, Sarvesh B, et al. The impact of NBTI on the performance of combinational and sequential circuits. In: Proceedings of the 44th ACM/IEEE Design Automation Conference, San Diego, 2007. 364–369
11 Borkar S, Karnik T, Narendra S, et al. Parameter variations and impact on circuits and microarchitecture. In: Proceedings of Design Automation Conference, Anaheim, 2003. 338–342
12 Chen G, Chuah K Y, Li M F, et al. Dynamic NBTI of PMOS transistors and its impact on device lifetime. In: Proceedings of the 41st Annual IEEE International Reliability Physics Symposium, Dallas, 2003. 196–202
13 Zhao W, Liu F, Agarwal K, et al. Rigorous extraction of process variations for 65-nm CMOS design. IEEE Trans Semicond Manufact, 2009, 22: 196–203
14 Xiang D, Zhang Y. Cost-effective power-aware core testing in NoCs based on a new unicast-based multicast scheme. IEEE Trans Comput-Aided Des Integr Circuits Syst, 2011, 30: 135–147
15 Xiang D, Chakrabarty K, Fujiwara H. A unified test and fault-tolerant multicast solution for network-on-chip designs. In: Proceedings of IEEE International Test Conference (ITC), Fort Worth, 2016. 1–9
16 Xiang D, Sui W, Yin B, et al. Compact test generation with an influence input measure for launch-on-capture transition fault testing. IEEE Trans VLSI Syst, 2014, 22: 1968–1979

17 Ferhani F, Saxena N, McCluskey E, et al. How many test patterns are useless. In: Proceedings of the 26th IEEE VLSI Test Symposium, San Diego, 2008. 23–28
18 Wang N J, Patel S J. ReStore: symptom-based soft error detection in microprocessors. IEEE Trans Dependable Secure Comput, 2006, 3: 188–201
19 Aitken R. Yield learning perspectives. IEEE Des Test Comput, 2012, 29: 59–62
20 Powell M D, Biswas A, Gupta S, et al. Architectural core salvaging in a multi-core processor for hard-error tolerance. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, Austin, 2009. 93–104
21 Eyerman S, Eeckhout L, Karkhanis T, et al. A top-down approach to architecting CPI component performance counters. IEEE Micro, 2007, 27: 84–93
22 Tschanz J, Bowman K, Lu S, et al. A 45 nm resilient and adaptive microprocessor core for dynamic variation tolerance. In: Proceedings of IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), San Francisco, 2010. 282–283
23 Petrica P, Izraelevitz A, Albonesi D, et al. Flicker: a dynamically adaptive architecture for power limited multicore systems. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, Tel-Aviv, 2013. 13–23
24 Carlson T, Heirman W, Eeckhout L. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Seattle, 2011. 1–12
25 Miller J, Kasture H, Kurian G, et al. Graphite: a distributed parallel simulator for multicores. In: Proceedings of IEEE 16th International Symposium on High Performance Computer Architecture (HPCA), Bangalore, 2010. 1–12
26 Kohler A, Schley G, Radetzki M. Fault tolerant network on chip switching with graceful performance degradation. IEEE Trans Comput-Aided Des Integr Circuits Syst, 2010, 29: 883–896
27 Gizopoulos D, Psarakis M, Adve S, et al. Architectures for online error detection and recovery in multicore processors. In: Proceedings of Design, Automation and Test in Europe, Grenoble, 2011. 1–6
28 Alizadeh B, Fujita M. A debugging method for repairing post-silicon bugs of high performance processors in the fields. In: Proceedings of International Conference on Field-Programmable Technology, Beijing, 2010. 328–331
29 Chang C-W, Chou H-Z, Chang K-H, et al. Constraint generation for software-based post-silicon bug masking with scalable resynthesis technique for constraint optimization. In: Proceedings of the 12th International Symposium on Quality Electronic Design, Santa Clara, 2011. 174–181