Cardio: CMP Adaptation for Reliability through Dynamic Introspective Operation

Andrea Pellegrini, *Member, IEEE,* and Valeria Bertacco, *Senior Member, IEEE*

***Abstract*—A modern digital system includes in a single chip many components: processing cores, large caches, memory controllers, and hardware accelerators. Looking forward, future semiconductor technologies will enable even higher device integration, overall increasing system performance while reducing energy consumption. Unfortunately, prominent experts agree that such technologies will be prone to both permanent and transient faults within their lifetime. With the goal of addressing this issue, we propose Cardio: a low-cost architecture for reliable chip multiprocessors. Our solution is based on a novel hardware/software co-design where silicon failures are detected in hardware and system reconfiguration is managed in software. Comparing Cardio with a state-of-the-art hardware-based resiliency solution, Immunet, we found that our design can achieve a comparable fault response time while requiring a much lower area overhead. The proposed solution relies on a distributed resource manager to collect information about a CMP component's health, and leverages a synchronized distributed control mechanism to recover from permanent failures. Such architecture can operate as long as at least one general-purpose processor is still functional. Our experimental evaluation indicates that the overall performance impact of Cardio is as low as 4.5%, and its dynamic reconfiguration time upon fault detection is comprised between 20 and 50 thousand cycles.**

***Index Terms*—Hardware reliability; modeling techniques; multiprocessor systems; reliability, availability, and serviceability; reliability, testing, and fault-tolerance.**

I. Introduction

CURRENT digital systems are extremely sophisticated: today's technology allows the integration of billions of transistors in a single chip. It is now possible to develop chip multiprocessors (CMPs) composed of numerous processors, large memories, and dedicated accelerators [1]–[4]. Typically, these components are connected via high-bandwidth interconnect networks. The benefits of these systems are so widespread that CMPs have successfully conquered various markets, from high-end servers to low-power mobile devices. Looking forward, transistor density is expected to grow even further as silicon fabrication process improves. Digital designs can take advantage of future technology advancements to deliver even better performance at lower cost and power consumption.

Unfortunately, prominent experts agree that further shrinks in transistor's size will severely degrade overall system reliability. Transistor devices will be increasingly more susceptible to both transient and permanent failures and this will lead to higher rates of manufacturing defects and runtime failures [5], [6]. Runtime transistor failures can be caused by a plethora of physical phenomena, such as: electromigration, gate oxide breakdown, negative-bias temperature instability (NBTI), and hot carrier injection. As transistor's size keeps shrinking, the negative effects of these physical phenomena worsen. It has been experimentally demonstrated that runtime permanent hardware defects can be extremely dangerous. Indeed, fault injections on detailed gate-level models reported that undetected faults—especially in functional units such as multipliers, floating point units, and dividers—are very likely to silently corrupt program output [7]–[10].

Thankfully, the extreme device integration that leads to higher fault rates also provides solutions to improve system robustness. Indeed, modern CMP architectures can isolate faulty processors without compromising the rest of the system [11], [12]. This characteristic has been leveraged to boost manufacturing yield; still, as of today, most computer chips do not tolerate runtime failures. Hence, modern CMPs cannot overcome the two most critical consequences of hardware failures: service disruption and silent data corruption.

Our proposed solution, called Cardio, is a low cost, distributed, hardware/software technique to manage CMP availability at runtime. Cardio relies on distributed and low-cost hardware detectors to run periodic tests on a CMP's components, while it delegates system's reconfiguration (in case of a fault occurrence) to software routines. Despite Cardio's extremely low cost, we show that it is effective and responsive in providing system-level fault detection and reconfiguration. In Cardio, each component (cores and interconnect) periodically tests its own circuitry and then broadcasts the test outcome to the entire system. A software resource manager collects these diagnostic notifications. If any component reports a problem, the resource manager dynamically recovers and reconfigures the system to work around the faulty hardware. Compared against Immunet [13], a solution based solely on hardware mechanisms, our hybrid approach is more versatile, more scalable, requires less area, and provides comparable response time to runtime failures. On one hand, hardware-only resilient techniques, such as Immunet, can achieve short response time at near-zero performance overhead. On the other hand, these solutions typically incur high area overhead and have limited scalability and flexibility. Cardio overcomes these limitations

Manuscript received September 26, 2012; revised May 10, 2013 and August 22, 2013; accepted August 30, 2013. Date of current version January 16, 2014. This work was supported in part by the NSF Grant 0746425 and in part by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. A portion of this work was published in the IEEE International High Level Design Validation and Test Workshop (HLDVT 2010). This paper was recommended by Associate Editor Y. Xie.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: apellegrini@umich.edu; valeria@umich.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2013.2284008

by executing in software the two most demanding tasks required in a resilient design: system-level monitoring and hardware reconfiguration.

Our solution relies on a recovery mechanism based on full system checkpointing that intervenes shortly after a fault detection, so as to minimize system down-time and storage requirements. Additionally, both hardware detectors and reconfiguration mechanisms are completely distributed, so that a Cardio-enhanced CMP does not present a single point of failure. Finally, our solution is independent from the interconnect topology and can enable reliable performance on any multiprocessor system. These characteristics empower Cardio to dynamically overcome hardware failures, hence extending a system's lifespan while reducing its overall operating costs.

In summary, this paper makes the following contributions to the area of online fault recovery and reconfiguration.

1) We introduce a novel distributed resource manager to overcome runtime faults in a CMP's cores and specialized hardware units. Hardware units periodically exchange diagnostic messages reporting their fault-free or faulty state. Distributed software routines execute on the general-purpose cores, collecting diagnostic messages and updating the system's availability map accordingly. We develop a complete resource discovery process to accomplish this goal, thus not requiring any a-priori knowledge of the system.
2) We propose a novel reliable routing solution for networks-on-chip. In contrast with other fault-adaptive routing solutions, Cardio relies on hardware fault detectors and on software algorithms for computing packets' routes. Upon fault detection, diagnostic messages are used to discover the new network topology and a distributed, software-based resource manager computes the new communication routes.
3) We provide a formal basis for Cardio's reconfiguration algorithm, showing that it is both livelock and deadlock free. We also compare Cardio against a complete hardware solution and find that our hybrid approach not only has a lower hardware footprint, but also provides much better scalability. Furthermore, as the number of cores in CMPs increases, Cardio's response time is constant with only a polynomial increase in its communication overhead.
4) Finally, we demonstrate that Cardio can greatly extend the lifetime of CMP systems, allowing defective chips to still deliver competitive performance. As the number of permanent failures in a silicon chip increases, we found that the performance of a Cardio-augmented system decreases gracefully. We also experimentally show that our approach is deployable on a variety of CMP topologies. Finally, we evaluate the overall cost of our new design on a complete and reliable CMP system.

II. Related Work

In this section we provide an overview of previous hardware and software solutions that aim at improving runtime reliability in CMPs, processors, and interconnect subsystems.

CMP reliability—Zajac *et al.* [14] propose a solution for CMPs comprising hundreds of tiles, each one composed of a single core and an interconnect router. Software tasks are mapped to tiles by special hardware units, called input/output ports (IOPs). IOPs are in charge of monitoring and scheduling jobs on functional tiles. While this approach allows dynamic hardware resource discovery, it also incurs in two major drawbacks. First, IOPs are dedicated hardware components that introduce a single-point-of-failure in the system. Second, reconfigurable components in this solution are tiles composed of both computational and communication elements and a partially faulty tile must be completely deactivated. In contrast, Cardio's discovery procedures discriminate between processors and interconnect components, hence maximizing hardware utilization.

Fault-tolerant microprocessors—Mission critical and high-availability computers rely on coarse grain component redundancy to improve reliability. Systems such as the HP NonStop and the IBM zSeries [15], [16] implement dual and triple modular redundancy, thus always incurring extremely high area, power and performance overheads.

Recently, several research projects have proposed low-cost reliable processor designs. The Bulletproof CPU targets specifically VLIW processors [17]. Cardio and Bulletproof share the approach of testing hardware components in the background, while concurrently computing speculative results within each computational epoch. Other works on reliable CPUs target components with natural redundancy such as the reorder buffer, the branch history table, caches, and other arrays of regular structures [17]–[19]. StageNet and Viper are more radical approaches to hardware reliability that develop reconfigurable interconnect fabrics connecting multiple hardware modules within a multicore system [20], [21]. In these two designs, hardware components (pipeline stages in StageNet and functional units in Viper) can be swapped to maintain performance even in the face of defective hardware. Cardio is orthogonal to and compatible with all these designs, since it focuses on overall system availability to overcome the challenges posed by runtime failures in CMPs. When deployed with the solutions aforementioned, Cardio would only intervene once they have exhausted their ability to overcome faults within individual processor cores.

Reliability in NoCs—Several solutions target reliable interconnects, including NoC routers capable of tackling runtime failures [22], [23]. However, they usually require high hardware overhead. Adaptive routing algorithms for NoCs are typically applicable only to simple topologies (such as meshes and tori) and cannot be extended to irregular ones. Cardio, on the other hand, is a low cost solution agnostic to both topology and routing algorithm.

An example of a distributed routing algorithm for reliable NoCs is Immunet [13], a hardware-based solution that can quickly recover from hardware faults (roughly ten thousand cycles for an 8x8 mesh). Upon fault detection, Immunet floods the network with diagnostic messages, and the number of messages exchanged grows exponentially with the number of nodes in the network. Since each router in the system updates its routing tables based on the messages received, the reactivity of these hardware solutions decreases exponentially as the system's size increases. In contrast, Cardio is somewhat slower in reacting to an interconnect failure, requiring between 20 and 50 thousand cycles for a full recovery, but its routers need limited hardware additions, and the number of diagnostic messages exchanged grows polynomially with the core count.

Moreover, since Cardio manages network reconfiguration in software, it enables the use of sophisticated routing schemes without increasing NoC components' complexity. The difference between Cardio and other static routing mechanisms is that our solution can periodically adapt messages' paths to the dynamic characteristics of the system. Furthermore, it provides a system-level knowledge of the CMP, hence enabling application-aware packet route tuning. Neither of these capabilities is achievable by current hardware-only solutions.

Finally, we should mention stochastic routing and smart-flooding: very low cost solutions for reliable on-chip communication, which unfortunately are only viable for lightly loaded networks [24], [25].

Reliability via middleware—Bressoud *et al.* [26] first investigated the adoption of a middleware layer to support hardware reliability. Their work provides a high cost reliability solution by replicating software execution. More recently, middleware-based reliability solutions were also proposed to tackle intermittent faults [27]. In contrast, Cardio's hybrid hardware-software mechanism utilizes hardware detectors to quickly identify permanent failures and relies upon software routines to reconfigure the system around them.

III. Cardio Architecture

Cardio follows a notification-reaction paradigm suited to improve the reliability of future CMPs. This section first reviews the fault models targeted by our work and then overviews the principles driving our design choices.

A. Hardware Failures Addressed

This paper targets permanent failures manifesting during the lifetime of a CMP and located in processor cores, interconnect routers and links in the intrachip communication subsystem. Hardware failures of such nature are caused, for instance, by the wear-out of a chip's physical components.

A first solution to handle hardware failures at runtime is to deploy an on-line failure prediction system [28], [29]. Such mechanism uses special circuitry within the chip to monitor system parameters degradation over time. The information collected from these monitors is then used to estimate the probability of failures in the near future. Unfortunately, this solution typically requires a large silicon real estate and long computations to extrapolate an accurate prediction.

A second technique proposed to tackle runtime failures consists of detecting hardware failures shortly after they manifest. Here, hardware components are tested at regular intervals in order to detect faults such as stuck-at, bridge or path-delay [30]–[32]. In these systems, execution is partitioned into computational epochs, and workload execution is periodically suspended to test the underlying hardware components. Since hardware failures may corrupt some computations, program state is backed up before executing a new epoch through a checkpointing mechanism. If the online tests succeed, processor logic is deemed as fault-free, the results from the previous epoch are committed, and a new checkpoint is created from the current program state. Otherwise, the system is reconfigured to isolate the faulty component, and system state is restored to the previous safe checkpoint.

Cardio adopts this second execution paradigm for reliable computing, since it has been shown to be both very economical, as low as 1% hardware overhead [32], and effective in achieving high fault coverage [30]. Hence, this paper targets runtime hardware failures that can be detected by such periodic checks. Our failure model assumes that a hardware component deemed faulty—for instance, because of a stuck-at fault in a microprocessor or in a router—can be disabled and will no longer provide any information about its state to the rest of the system.

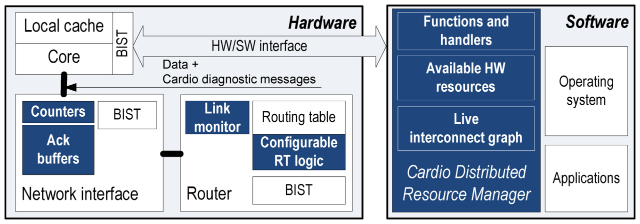

Fig. 1. Cardio architecture overview. Cardio hardware and software additions are highlighted in the figure. Communication endpoints are augmented with acknowledgment buffers and counters to determine transmission failures. Routers are enhanced with logic to diagnose link-connectivity and to reconfigure routing tables. Each general-purpose core in the system executes an instance of the distributed manager.

This type of fault-tolerant designs must face the issue that hardware failures may also occur in their self-test logic. Since Cardio's self-testing logic is local to each component, we assume that faults in this circuitry cause the corresponding component to be non-testable and therefore detected as faulty. Because the self-test circuitry is activated much less frequently than the logic in the rest of the system, its vulnerability to wear-out is significantly lower [23]. However, Cardio relies heavily on these detection mechanisms to check hardware integrity, hence, it may be worth protecting the self-testing logic with traditional reliability mechanisms such as dual-modular redundancy.

Finally, because Cardio's hardware self-discovery operations might sometimes interrupt intrachip communications, our solution addresses temporary communication glitches through end-to-end message retransmission. Thanks to this mechanism, Cardio can overcome communication problems due to transient or intermittent faults. Lastly, corrupted messages that need to be retransmitted can be promptly recognized through low-cost error detection codes [33].

B. Design Philosophy

In order to ensure correct operations on a CMP subjected to permanent hardware failures, we need to address two problems. First, all hardware components must be diagnosed as either functional or unavailable. Second, connectivity and possible communication paths among functional components must be determined. Current solutions for runtime on-chip reliability typically rely solely on expensive hardware mechanisms to achieve both these goals. Furthermore, no previous research provides a complete solution for distributing and managing components' diagnostic information in CMPs.

In Cardio, each hardware component is equipped with a self-test feature that periodically broadcasts diagnostic test outcome to the entire system. Differently from prior works, a distributed resource manager executing on each of the general-purpose cores dynamically maintains and organizes information about the CMP's hardware units. Fig. 1 shows a high level schematic of the hardware and software additions necessary to equip a baseline CMP system with the proposed Cardio features.

In order to maintain a low area overhead, hardware additions are limited to the intrachip communication subsystem and consist of enhancements to both network interfaces and routers. Network interfaces are augmented with: 1) a buffer to store transmitted packets waiting for acknowledgment, and 2) a set of counters to trigger automatic retransmission in case of time-out. Routers are enhanced with: 1) a link monitor to collect information about the components directly connected to the router, and 2) a configurable routing table to direct NoC packets to their destination. Cardio software additions are more significant, and consist of: 1) a data structure to contain information about the CMP's state (list of cores available); 2) a graph of the connectivity among the functional on-chip components; and 3) the software routines necessary to handle diagnostic messages and reconfigure the hardware system.

Lastly, cores are augmented with hardware structures that enable program checkpointing. While local data caches can use low-cost solutions, such as versioning, to store data belonging to different checkpoints, each processor must include a shadow register file to store the values computed in the previous epoch [34], [35].

IV. Cardio Runtime Operation

In a Cardio-equipped system, execution time is partitioned into epochs. The system cannot guarantee the correctness of its computed results until the underlying hardware has been tested. Therefore, all results are considered speculative until hardware tests assess that no fault occurred during the previous epoch. Furthermore, in order to overcome glitches that might affect intrachip data transmissions, messages are temporarily buffered to allow end-to-end packet retransmission.

Because resource manager instances execute independently on the CMP's cores, they must organize so they can share an identical estimate of hardware components and, if needed, can enforce a sound system reconfiguration. The problem of reaching a common decision among several components is a simpler instance of the Byzantine Generals' Problem [36], called the consensus problem [37]. Solving this problem in our case consists of providing a common knowledge of the available resources to all non-faulty cores in the CMP (this is also known as the consensus vector). Cardio relies on the periodic broadcast of diagnostic messages to provide an efficient solution to this problem. The remainder of this section details the mechanisms leveraged to achieve this goal.

While an application is speculatively executing an epoch, processors and NoC routers periodically and independently suspend their tasks to test the integrity of the underlying hardware. These local tests are not globally synchronized, and the only constraint imposed by Cardio is that all hardware units must complete their self-tests by the end of the current epoch—so that the previous one can be safely committed.

After each self-test completes, its outcome is broadcast to the rest of the system. With the goal of minimizing performance impact, each unit shares only the necessary diagnostic information with the rest of the system. For instance, interconnect connectivity is tested every few thousands cycles: if each router were to broadcast test outcomes so frequently, these messages would severely burden the entire communication infrastructure. Therefore, routers broadcast system-wide updates only upon the discovery of a new hardware failure. In our design, more frequent diagnostic tests lead to more prompt reactions to failures, but also entail higher performance impacts and diagnostic message proliferation. Indeed, diagnostic frequency is a design trade-off that we analyze in Section V. Since Cardio's objective is to dynamically react to detected permanent hardware faults, we rely on previously proposed techniques to diagnose faulty hardware components, such as those presented in [22], [32], and [38].

Diagnostic messages are collected by the various instances of the distributed software manager. Each of them, independently, updates two local data structures: the list of available hardware resources and the graph of the functional interconnect links (shown in the right side of Fig. 1). The first structure lists all hardware resources still available in the CMP, while the second is used to compute the routes of NoC packets. When a resource manager detects a new hardware failure, all speculative computations since the end of the last committed epoch are discarded. The system is then reconfigured to work around the fault. Otherwise, if no hardware fault is detected, all resource managers commit the results produced in the previous speculative epoch and restart program execution. Note that, after a hardware failure, the state of each active component in the CMP must be recovered to restart software execution. For this purpose, Cardio can rely on either software or hardware checkpoint techniques [34], [35].

As we demonstrate in this paper, a CMP can rely on inexpensive diagnostic message broadcasts to promptly react to system alterations without significantly hindering its performance. Hence, the design principles developed in Cardio are extensible to a variety of digital systems, as long as at least one general-purpose core is available to execute an instance of the resource manager. Furthermore, Cardio can also be deployed to enable hardware to adapt its functioning to network traffic, components usage, and temperature. For instance, each router could provide information about the usage of its links. Routers experiencing high utilization can then broadcast this information to the distributed resource manager, so it can reroute packets to mitigate traffic congestion.

Since our hardware availability assessment and reconfiguration procedures differ for processor cores and NoC components, the next two sections discuss them separately.

A. Cores

Handling runtime failures on processors requires several steps. As soon as the self-tests detect a hardware problem in a processor, the system must suspend its execution and reconfigure to prevent faulty hardware from corrupting software output. In order to gather and distribute system-level information about CMP's cores integrity, all processors in a Cardio-enabled design follow the sequence of operations illustrated in Fig. 2.

Cores in modern CMPs rely on independent clock signals, which cannot be easily synchronized to provide a global signal to all components in the system. Therefore, each core asynchronously and periodically suspends its normal execution to perform self-tests (step 2 in Fig. 2). These intervals are typically several tens of million cycles long—computational

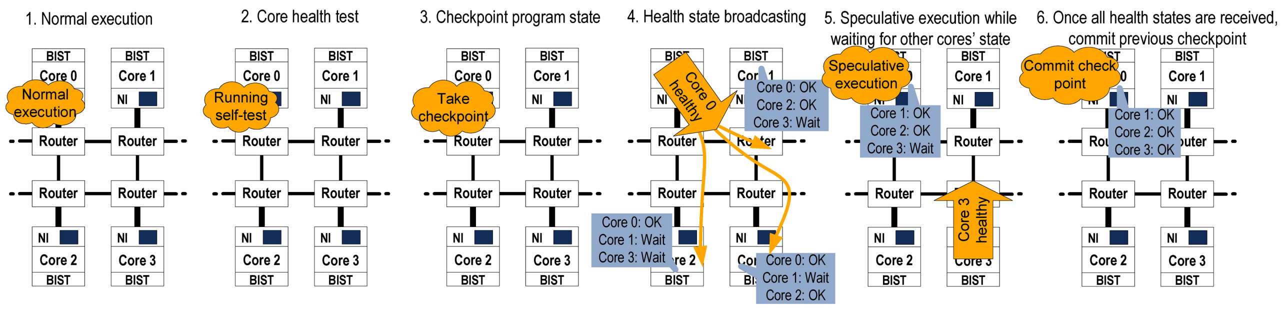

Fig. 2. Core monitoring and recovery in Cardio. To maintain an up-to-date state of the available cores in the system, Cardio relies on the following five step sequence. 1) The cores perform their normal functions. 2) Core 0, independently from the other cores, executes a self-test procedure to detect potential permanent failures. 3) If the test completes successfully, a local checkpoint of the current core state is taken. 4) A diagnostic message is broadcasted to all other cores to signal that core 0 is functional. 5) Before core 0 can commit its computation, it must receive successful fault-free acknowledgments from all cores that were functional in the just completed epoch. In the meantime, it can speculatively continue its execution. 6) Finally core 0 receives the last positive fault-free acknowledgment from core 3 and commits its results up to the last checkpoint.

epoch lengths adopted in previous works are 10M, 20M, 100M, and 1 000M cycles. Techniques ranging from structural to functional testing have been proposed to perform online checks of digital designs [30], [32]. Periodic testing may decrease overall core performance by 1% to 30%, depending on the workload, testing technique, and epoch length [30], [32], [39]. Furthermore, briefly pausing cores might cause jitters in the execution of an application. Thankfully, both these disadvantages can be significantly mitigated through modifications to the operating system scheduler [40].

If a processor successfully passes all self-tests, its architectural state and memory state are checkpointed (step 3 in Fig. 2). Test results are then wrapped in a diagnostic message marked with the unique identifier of the tested core, and broadcasted to the entire system (step 4 in Fig. 2). Depending on the granularity of the test, more detailed information about the impact of the faults affecting a core can be provided. For instance, it may be possible to report a core with a non-functional floating point unit as available, but only capable of executing integer instructions [32].

Since all resource managers must agree on the available hardware resources, Cardio imposes a barrier to allow all cores to synchronize their information about the state of the system. While waiting to receive diagnostic messages from all other processors, a core may speculatively start executing the subsequent epoch. For instance, step 5 of Fig. 2 shows core 0 starting a new computational epoch while still waiting for a diagnostic message from core 3. Only when all diagnostic messages from the functional cores are received, a local checkpoint is safely committed (step 6 of Fig. 2). If n is the number of healthy cores in the CMP, a core may receive at most $n-1$ unique diagnostic messages from other cores for each epoch.

Speculative output and program state computed in an epoch and stored in a checkpoint are committed when all active processors in the CMP agree that no failure occurred during that epoch. To manage speculative memory state, we opted for a solution inspired by ReVive [34], where each node only logs the content of the cache lines that are modified during an epoch. Cardio mechanisms provide a system-level synchronization primitive that enables the deployment of a low-cost global checkpointing system.

When a hardware failure is detected, all computations performed since the previous checkpoint cannot be trusted, since such fault might have corrupted software execution. Therefore, all speculative results are discarded and the faulty hardware is identified and disabled.

A resource manager detecting a new hardware failure sends a special message to all cores to rollback to the previously synchronized checkpoint, so as to prevent the commitment of potentially corrupted speculative results. A faulty core is not required to report it faulty state to the rest of the system: other instances of the resource manager will detect a missing diagnostic message at the end of their speculative epoch by mean of a local timeout. Then, the defective component is disabled, and a reliability-aware operating system migrates the active applications to use only the available resources. Lastly, if part of the system becomes isolated due to a failure in the interconnect, the memory content of the isolated nodes or in failed memory controller can be retrieved through ad-hoc mechanisms, such as DRAIN [41].

In order to prevent potential livelocks and guarantee that all cores in a same connected region have a consistent view of the functional hardware, resource managers exchange checksums of the reconstructed hardware configuration. In case of checksum mismatch, for instance because of delays in the communication infrastructure, Cardio forces all cores to suspend their activity to drain all in-flight communications from the system. All cores then restart their diagnostic protocol and exchange a second set of checksums that summarize the state of the system. This second time both the diagnostic messages and the checksums are guaranteed to arrive in time, since the system is not burdened with any other traffic.

Cardio's diagnostic protocol is inspired by the one extensively discussed in [37], which is proven to be deadlock and livelock free if the four following conditions are met.

1) The communication system is reliable and only cores can be subject to failure.
2) Each core can determine the sender of any received message.
3) Any core's failure to send messages is detectable.
4) Any non-faulty core can broadcast information to all non-faulty cores.

Our system meets condition 1 because: 1) temporary communication failures are tackled through an end-to-end network-level retransmission protocol, and 2) permanent communication errors cause disconnected cores to be detected as faulty.

Condition 2 is fulfilled since each core marks all generated messages with its unique ID. Timeout counters enable the detection of lost packets and make Cardio meet condition 3. Finally, condition 4 holds true because Cardio only applies to interconnects supporting message broadcasting. Our protocol works as follows. At the beginning of a computational epoch:

1) each j of the r cores available is tested, and its diagnostic v_j message is broadcasted to the system (for instance, we can use a single bit flag set to "1" if a core is available and to "0" if faulty);
2) each instance of the resource manager:
   a) if it receives a value v_j from all cores, $1..r$, then it takes v as its system's availability image; broadcasts v (or its checksum) to the system; waits until the end of the computational epoch;
   b) otherwise, if it does not receive a message from at least one of the cores expected to be available (a timeout occurs), then it broadcasts such information to the system.

At the end of the epoch, each resource manager:

1) if it received a matching value v from all other cores during the epoch just completed, then it confirms v as the consensus vector and commits the results generated in the previous speculative epoch;
2) otherwise, a new fault has been detected; program state is rolled back to the previous safe checkpoint, and the system is reconfigured to disable the faulty core.

While the correct execution of this protocol has been proven to be both deadlock and livelock free [37], it is worth discussing the case of two byzantine faults that may disrupt the behavior of this procedure. First, faults may silently corrupt the data carried by a message used in Cardio's protocol. We protect our design from this event by augmenting all messages with error correcting codes [33]. The second case is due to faults in the self-test logic, which may cause a broken core to incorrectly advertise it status as available. This eventuality can be avoided by protecting the self-test logic with traditional reliability mechanisms such as dual-modular redundancy.

In order to maintain knowledge about the available hardware, each resource manager builds a list of available cores from the received diagnostic messages. Diagnostic messages need to be synchronized to enforce that all cores will receive them before the end of the next epoch. Since each core generates diagnostic message independently, Cardio relies on real-time counters to trigger the broadcast of diagnostic messages. Cardio's introspective operations rely heavily on such timers, and their hardware should be tested thoroughly and frequently or even duplicated to ensure their functionality. We send at least two copies of these messages within one epoch to guarantee their timely arrival: in fact, even if communication errors cause the loss of one message, the other is very likely to arrive on time. Note that it is still possible, although improbable, that both messages may get lost due to transient failures. This case is not critical since it only forces Cardio to flush all in-flight operations and initiate a new test routine.

B. Interconnect

Interconnect correctness and performance are fundamental for any CMP. On-chip routers deliver messages between cores, and a single router can connect multiple processors.

```
 1: Drain output links
 2: Test Router logic through BIST
 3: For each output link
 4:   Send discovery request
 5: For each output link until timeout
 6:   If discovery response received
 7:     Update link table
 8: If link table changed
 9:   Broadcast updated link table
10: Resume operations
```

Fig. 3. Router periodic test procedure. First, the online testing algorithm on the router tests the router's hardware. Then the state of the direct links between the router and its neighbors is checked. Directly connected neighbors that do not respond within a certain time threshold are considered not available. Note that only changes to the local link table are broadcast to the system.

Cardio can be successfully adopted in any NoC topology, since interconnect state and routing information are handled in software by the distributed resource manager. Previous work characterized NoC malfunctions as either: 1) corruptions in the payload/data, or 2) errors in the delivery system [42]. The first kind of errors can be easily addressed with error-correcting codes and retransmission. The latter can cause packet loss or network deadlock, and both behaviors can be directly mapped to malfunctions in a router's links. Here, we introduce a routing algorithm that dynamically discovers link failures in an arbitrary network and updates communication routes accordingly. In general, two families of routing algorithms are available for this purpose: link-state and distance-vector [43]. On one hand, self-configuring NoCs typically adopt some flavor of the distance-vector protocol, because its limited complexity well suits hardware implementations. On the other hand, algorithms based on the link-state protocol introduce lower communication overhead, therefore converge faster and scale better than those based on distance vector.

In typical link-state protocols, for instance the ones developed for computer networks, every node constructs a graph of its local network connections and broadcasts it to the others. Then, each node, independently, computes the best path from itself to every possible destination in the network. Hence, network nodes only exchange information about their local connectivity, but must perform complex computations to generate network routes. Cardio overcomes this drawback by delegating route generations to the software resource manager.

Fig. 3 shows the steps performed by each router when the online testing procedure is activated. First, each router independently suspends its activity to perform a self-check on its own hardware structures, for instance testing its input and output buffers and its crossbar (step 2 in Fig. 4). Any of the several techniques proposed in the literature can be adopted for this purpose [22], [23]. The outcome of this test determines whether the router is operational. Once this first check phase is completed, the router under test probes its links to discover all directly connected neighbors. Each node in the NoC performs local link discovery independently. To maintain an accurate state of the local connections, each router periodically generates a discovery "heart beat" that is sent to all adjacent nodes, as illustrated in step 3 of Fig. 4. A router receiving a heart beat discovery responds including its node ID (step 4 in the figure). Each link monitor then populates a

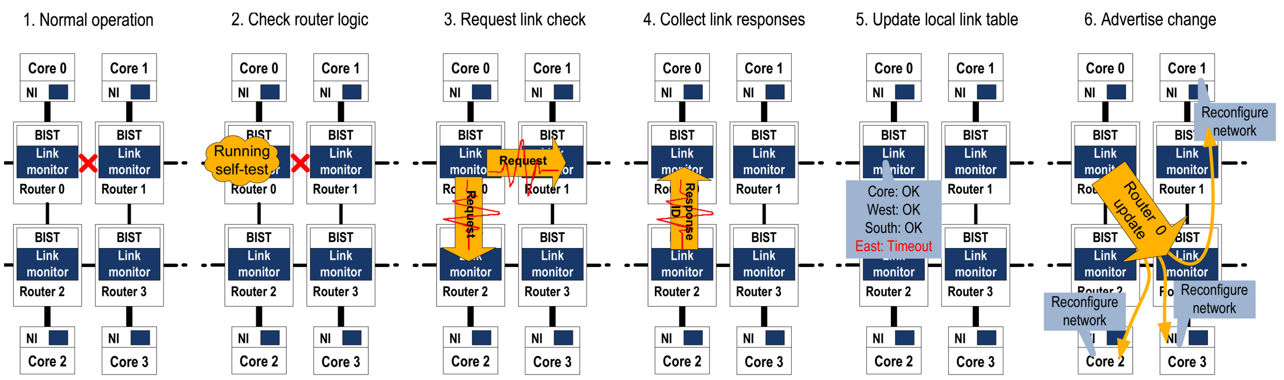

Fig. 4. Dynamic interconnect management in Cardio. Self-discovery and reconfiguration in the interconnect are organized in five steps. To reduce the amount of extra traffic in the CMP, only topology changes are advertised. In the figure: 1) the NoC performs its normal functions; 2) router 0 suspends its execution to perform a self-test routine; 3) since its hardware is found to be functional, discovery messages are sent to all output links; 4) router 2 replies to the request with its ID; 5) since no response is received from router 1 within the deadline imposed by the timeout, the failed link is detected; and 6) because of the new fault, router 0 broadcasts a diagnostic update requesting to reconfigure the network.

table where every functional local link is associated with the ID of the node connected to it (step 5 in the figure). Routers also store the unique identifiers of all directly connected cores.

Routers may trigger the detailed hardware tests performed in the first step rather infrequently, since accurately testing NoC hardware components often requires a considerable amount of time. Indeed, other directly connected routers can discover critical failures that jeopardize a router functionality, and failures causing packet corruption can be detected by error-checking codes.

After these two phases, each router is able to detect failures that prevent communication with its directly connected nodes, perhaps because a failure interrupts a communication path. Once a router discovers a new link malfunction, it discards all packets directed toward the broken link. The updated local link table is then broadcasted to the system to notify the resource managers about the change in network topology (step 6 in Fig. 4). Due to storage and performance constraints, the period between two subsequent link tests is limited to a few thousand cycles, as discussed in Section V-C. All links, even those previously considered faulty, are periodically tested. This allows routers to recover parts of the network that are only temporarily unavailable, for example due to intermittent faults or high traffic congestion. When a failure is detected, the system starts the following hardware reconfiguration routine.

1) The router that discovered the failure broadcasts this event to the entire system.
2) Resource managers receive this notification, pause the cores, and discard all speculative computations.
3) Software routines are triggered to compute new routes and the hardware is reconfigured accordingly.
4) The system rolls back to a previous checkpoint to restore software state and restart execution.

Since our dynamic network testing routine might discard in-flight packets, we deploy a retransmission mechanism to avoid communication loss. Cardio addresses sporadic transmission glitches through an end-to-end acknowledgment protocol: every time a message successfully reaches its destination, the receiver notifies the sender. All interconnect endpoints therefore are enhanced to maintain hardware counters and store pending messages waiting for acknowledgment. These counters are incremented every cycle, and any message that does not receive an acknowledgment within a certain time threshold triggers a timeout. In case of timeout, the network interface retransmits the timed out message; if the second attempt is also unsuccessful, the network interface affected by this problems notifies its directly connected cores of a potentially more severe reliability threat by raising a hardware exception. Acknowledgments may be sent through specialized packets or may be piggybacked to regular data packets. Acknowledgment buffer size is a storage and performance trade-off, which is evaluated in Section V.

C. Cardio Distributed Resource Manager

Cardio's distributed resource manager is responsible for monitoring and managing the system's reconfigurable hardware. This light-weight software layer leverages the information collected from the local tests to assess hardware availability and connectivity. When necessary, Cardio suspends user applications running on a core to execute resource manager's maintenance routines. User applications are also interrupted every time a hardware component raises an exception or when a diagnostic message is received.

Resource managers use the information about local connections broadcasted by the routers to generate a connectivity map of the on-chip network. All cores in a connected region reconstruct the same topology. If the interconnect is partitioned into multiple disconnected regions, each core running an instance of Cardio's resource manager only reconstructs the region to which it belongs. Once the interconnect graph is built, one local resource manager (for instance the one running on the core with the lowest identifier number) computes and distributes all routing tables, thus configuring the system to permit communication among all available components. From the diagnostic messages broadcasted to the system each resource manager also populates the list of the available hardware components and the interconnect graph. As previously discussed, a checksum of these two data structures is transmitted to all cores in the CMP to verify that all resource manager instances agree on the current state of the system. If a checksum mismatch is detected, resource managers initiate a renegotiation among themselves, eventually pruning routes not accessible by one or more cores.

V. Experimental Evaluation

Cardio proposes a distributed mechanism to manage and organize on-chip resources at runtime. Therefore, it adds extra on-chip traffic due to the diagnostic messages exchanged by the self-checking hardware components. Thus, our experiments first evaluate Cardio's impact on the system's interconnect, considering a variety of topologies and workloads.

We initially study the optimal size of the acknowledgment buffers at the NoC endpoints and measure packet latency sensitivity to interconnect discovery. We then evaluate the behavior of Cardio on a system affected by hardware failures. We first analyze the effect of hardware failures on the performance of a system in steady state, where faults are injected before starting our simulations. We then evaluate our design when subjected to runtime hardware failures. We also report the impact of our solution on interconnect performance and energy. Finally, we conclude our study evaluating the performance and area impact of all mechanisms required to deploy a fault-tolerant CMP: software state checkpointing, core self-testing, and Cardio.

A. Experimental Setup

We used two different simulators for our experiments. The first one is a fault-aware system-level C++-based simulator. We adopted this infrastructure to evaluate Cardio's effects on interconnect performance and measure its response time to hardware failures. Since our target is to explore the effectiveness of Cardio's protocols, we developed a simulator that allows quick turn-around for a range of architectures and HW/SW co-design. To this end, our simulation includes models at different levels of abstraction: cores' behavior is implemented at the transaction-level through clock counters, while the interconnect model is cycle-accurate at the packet granularity (we do not consider flit-level structures nor virtual channels). We validated our in-house simulator by comparing the average packet latency of comparable fault-free systems in steady-state mode with results produced by other available NoC simulators [38], [44]. The second simulator, gem5, was also used to estimate the impact on performance of a complete reliable system deploying Cardio [45].

In order to measure Cardio's response to hardware failures, our simulator models faults in the interconnect links. Our fault models target the two following faulty behaviors: 1) packets dropped before reaching their destination, and 2) packets stuck at some network node [42]. In the first behavior all packets attempting to traverse a broken link are dropped (drop-packets), while in the second behavior a communication path is blocked at a faulty link (hold-packets). These two types of faults represent the worst-case scenarios observed in previous evaluations with RTL models [23].

The CMP simulated with the first simulator is composed of 16 cores, each connected to a dedicated network interface. We consider four different interconnect topologies: ring, mesh, torus and crossbar. The system frequency is set at 2.4GHz, with five-stage routers transferring packets of up to 32 bytes in size. Packets are buffered at every router; routers can store up to two packets at the time. In our experimental evaluation we adopted source routing, embedding routing information in the packet itself. Routing tables are stored in the network interfaces and communication paths are computed by the resource manager using the up*/down* routing algorithm [46]. Cardio does not impose limitations on the routing algorithm adopted:

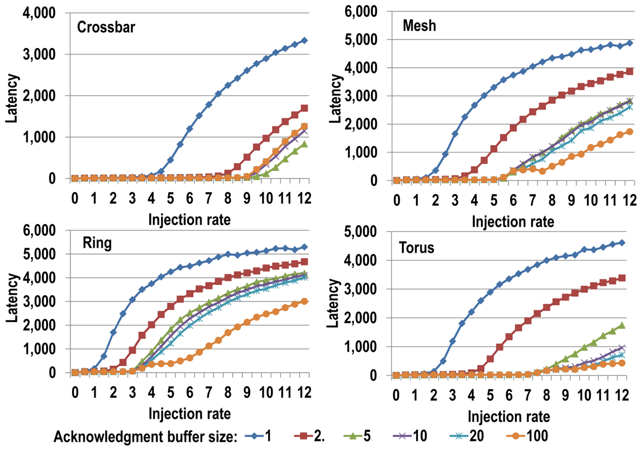

Fig. 5. Packet latency versus injection rate for different acknowledgment buffer sizes. Each curve represents a different acknowledgment buffer size as indicated in the legend. The x-axes represent each node probability (in %) of attempting to inject a new packet in the network. Buffers storing up to ten packets provide the best trade-off between storage and communication latency.

these design choices were driven by the goal of simplifying troubleshooting. In order to measure Cardio's impact on interconnect performance, we considered uniform random traffic as well as traces from the SPECMPI benchmark suite [47]. On one hand, random traffic ensures uniform link utilization so that packet latency and fault impacts are not biased by traffic patterns imposed by a benchmark's characteristics. On the other hand, traffic patterns from the SPECMPI benchmarks heavily stress intercore communication and thus provide a worst-case scenario for estimating Cardio's performance and traffic overhead. For the uniform random traffic injections we report packet injection rates as the probability that a core attempts to inject a new packet in the network (in %).

We then rely on the data collected through this first set of simulations to perform a full-system analysis of Cardio's impact on a complete, reliable CMP system. In this second set of experiments we measure the performance overhead of all the mechanisms needed to protect a hardware system against runtime failures: the checkpoint system, hardware self-test routines, and Cardio. In these experiments we also vary the size of the design under test to evaluate Cardio's scalability, and studied 16 different CMP configurations, ranging from designs containing 2 to 32 cores. Cores in these designs communicate through a network-on-chip organized in a mesh topology. We used the gem5 simulator in full-system mode to measure the performance impact of these designs [45]. For this second evaluation we assume a fault-free system and measure the impact of all necessary reliability mechanisms on the MEVBench benchmark suite [48]. Three main reasons led us to choose this set of benchmarks. First, its applications present a balanced amount of local computation and intercore communication. Second, it allows high degree of scalability. Third, this benchmark suite is representative of high-performance multithreaded workloads.

B. Acknowledgment Buffer Sizing

In order to handle communication failures, Cardio considers any point-to-point data transmission incomplete until the sender receives an acknowledgment from the receiver.

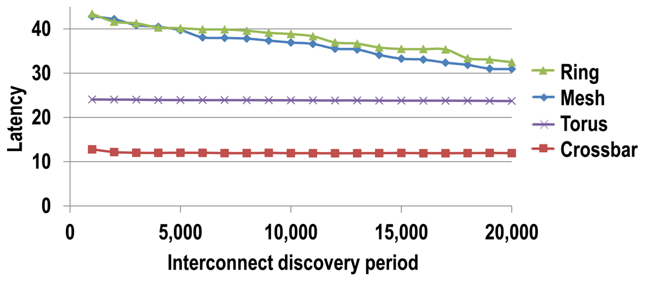

Fig. 6. Packet latency versus discovery period. Packet latency sensitivity to the discovery period differs for the analyzed topologies: mesh and ring are more sensitive to variations due to a smaller network bisection.

Thus, every non-broadcasted packet is temporarily stored in an acknowledgment buffer at the source until a confirmation message is received. The goal of our first experiment is to study the trade-off between storage and average traffic latency on network interfaces augmented by packet acknowledgment buffers. We evaluated several buffer sizes, ranging from 1 data packet (that is, the network interface must receive the acknowledgment for a previous packet before transmitting the following one), up to 100 outstanding packets. No faults are injected for this experiment. Fig. 5 shows the relation between the number of outstanding messages and the average packet latency. Traffic injection rate is measured as the probability of each network interface to attempt the injection of a new message in the interconnect at any given clock cycle, while packet latency is measured as the number of cycles between when a data packet is generated to when it is received by its destination. For the considered topologies we found that an acknowledgment buffer of ten packets is a reasonable compromise between storage requirements and packet latency. Indeed, acknowledgment buffers containing less than ten data packets lead to significantly worse average packet latency, while even doubling their size provides minimal benefits. Thus, network interfaces in all subsequent experiments include acknowledgment buffers capable of containing up to ten outstanding packets. Note that, the latency curves we observed level off as the traffic injection rate increases. Indeed, when the acknowledged buffers fill up, the cores are forced to suspend execution, creating de facto a self-throttling effect.

C. Dynamic Discovery Period

We then analyzed the effects on interconnect latency due to the extra traffic caused by the discovery packets. With this goal, we studied the sensitivity of average packet latency to the interconnect discovery period. This experiment was run with a traffic injection rate of 5% and a fault-free interconnect. From the data gathered in our analyses we observed that resource contention in the network starts to impact packet latency at injection rates higher than 5%. As the period between interconnect discoveries increases, average packet latency decreases due to bandwidth limitations. As shown in Fig. 6, this trend is steeper for topologies such as mesh and ring, for which links are subjected to a higher contention. Given the results obtained in this experiment, the network discovery frequency for our subsequent analyses is based on three different discovery periods, from a very frequent periodic test of 5 000 cycles to a much slower discovery period of 20 000 cycles.

D. Steady-State Faulty Behavior

Our third set of experiments evaluates how a faulty Cardio-enabled CMP performs in steady-state. For this experiments we rely on random traffic patterns. Faults are injected randomly in network links before starting each simulation. We perform these studies in order to show Cardio applicability to improve manufacturing yield and measure its performance in various faulty topologies.

Fig. 7 reports the packet average latency as a function of the number of faulty links in the system (x-axis) and the traffic injection rate (z-axis). As with the previous experiments, traffic injection rate is measured in probability (%) of packet injection per core. Interestingly, packet latency for the crossbar, ring, and mesh reduces as the number of faulty links increases. For the crossbar, this phenomenon is caused by the fact that a single faulty link is sufficient to disconnect a processor from the system. Therefore, as the number of faults increase, the number of active cores connected to the crossbar—and thus the traffic injected into the system—decreases. In the ring, hardware failures partition the topology in smaller, partially connected sub-networks. However, our graphs show that just a dozen faults suffice to disconnect most cores from the system, causing the average packet latency to plummet to zero. Fig. 7 also shows that both the mesh and the torus can tolerate a high number of hardware failures, maintaining performance up to and beyond 20 faulty links. This is due to the high connectivity of both these topologies, a characteristic that allows them to better work around hardware failures. Nevertheless, an increase in the number of faults in the mesh leads to a lower average packet latency. We found that interconnect failures partition the system into smaller sub-networks, which experience less average traffic congestion and thus lower packet latency. Other researchers have also reported this phenomenon [23]. In contrast, the torus does not manifest this behavior. In fact, the higher number of links available in this topology allows it to maintain connectivity among most nodes in the system, even when affected by more than ten faults. Nevertheless, faulty links do affect packets' possible routes, hence increasing the average communication latency. The observations above are confirmed empirically by our analysis of CMP connectivity as a function of the number of faulty links, whose results are shown in Fig. 8.

E. Runtime Faulty Behavior

In this section we studied the dynamic behavior of Cardio on a mesh at the time of occurrence of a permanent fault, evaluating its reactivity in detecting and overcoming a link failure. For this experiment, we simulated a fault-free system for 150 000 cycles to reach a steady state, when a randomly selected link is modeled as faulty. The two fault models aforementioned were considered for this paper: drop-packets and hold-packets. In order to provide insights on Cardio dynamic behavior, we analyzed the system at windows of 500 cycles and report, on the y-axis, the average latency incurred by all packets generated during each analyzed window. In this experiment we considered discovery periods of 5 000, 10 000, and 20 000 cycles. To stress the interconnect with a moderate amount of traffic, we set the packet injection rate at 5%. Through native execution profiling, we measured that the time required for the distributed resource manager to recompute the routing tables

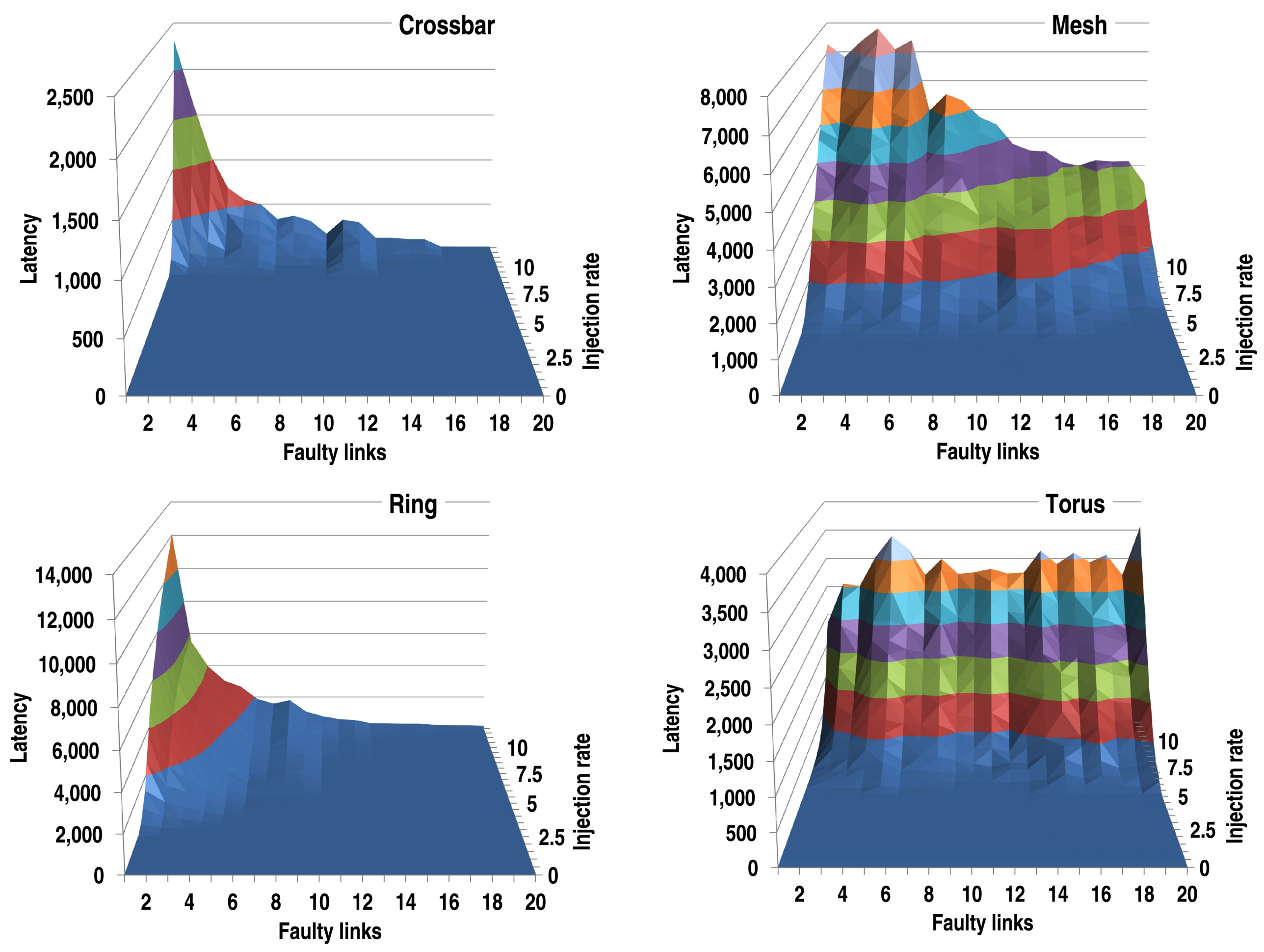

Fig. 7. Average packet latency for faulty topologies. These graphs report the packet average latency (*y*-axis) as a function of the number of faulty links in the system (*x*-axis) and the traffic injection rate in % (*Zz*-axis). When subject to faulty links, different topologies respond differently. Mesh and torus have higher connectivity and can maintain reasonable packet latency even when subjected to a significant number of faults.

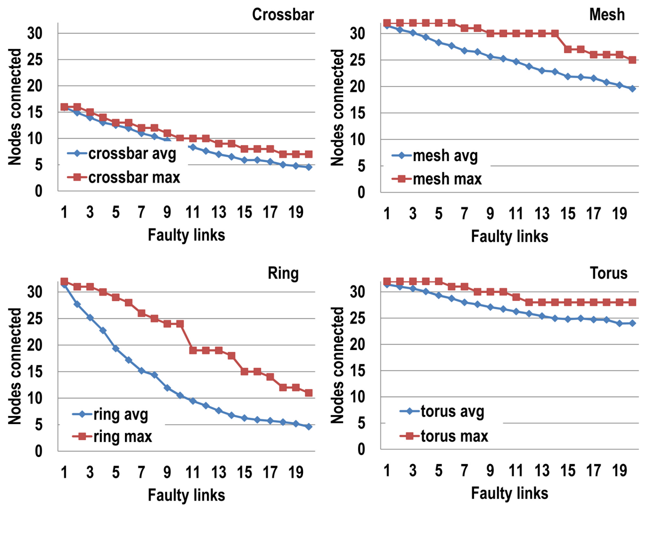

Fig. 8. Number of nodes connected versus faulty links. Our solution uses up*/down* routing to achieve maximum connectivity. The curve that characterizes the number of connected nodes depends on both the topology and on the number of available links.

is approximately constant at 10 000 cycles. We also estimated that each routing table requires 450 cycles to update, representing a serial write process for 15 routes of 15 hops each, writing 2 bits per hop [49]. In these experiments the network discovery period starts when the fault is injected to demonstrate the worst-case performance of our solution. In this first evaluation, we disregarded the extra traffic introduced by core diagnostic messages, since their transmission frequency is three orders of magnitude lower than for the interconnect components [22].

Results from the drop-packet fault model are reported in Fig. 9(a), where we distinguish a minimum of two and a maximum of three latency peaks, depending on the discovery period. The first peak is caused by the occurrence of the fault, and affects all packets that need to be retransmitted due to the faulty link. After a certain amount of time, directly related to the network discovery period, a first router detects the problem locally and consequently broadcasts the updated system state. The first interconnect reconfiguration process causes the network to temporarily stall, resulting in the second peak observable in the graph. The third peak shown in the graph is due to a second system reconfiguration and it is triggered by a later detection of the fault by a second router.

The impact of the hold-packets fault model is more dramatic: a fault's effect is not limited to packets in transit between two nodes, but rapidly propagates to a vast portion of the CMP, as demonstrated by the much higher average packet latency reported. Indeed, this second fault model congests multiple links: input and output buffers at the routers connected

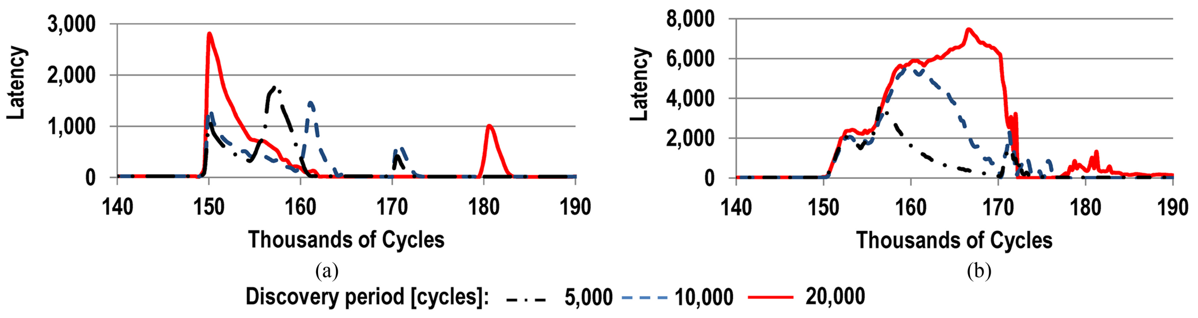

Fig. 9. Effect of a dynamic fault on a link. These graphs plot the average time necessary for a packet to reach its destination; packet latency is averaged between all packets generated in a window of 500 cycles. In this scenario, the link is broken at cycle 150 000 and two fault models are considered. (a) Drop-packet. (b) Hold-packet.

TABLE I
COMPARISON OF PERFORMANCE AND OVERHEAD OF CARDIO AGAINST A FULLY HARDWARE APPROACH, IMMUNET

		Cardio	**Immunet**
Fault detection		Hardware	Hardware
System reconfiguration		*Software*	Hardware
Topology		Arbitrary	Arbitrary
Router hardware overhead		6 bytes	28 bytes
Reconfiguration time	Torus 8x8	~50,000	~10,000
	Torus 16x16	~50,000	~36,000
Reconfiguration messages	Torus 8x8	1,032	12,240
	Torus 16x16	4,104	244,908

through the broken link fill up and cause a domino effect to their neighbors and then to the rest of the network. As reported in Fig. 9(b), the longer the period between hardware tests the more dramatic are the fault's effects on the overall system.

To put these results in context, we compare the dynamic behavior of Cardio with Immunet, a hardware solution for reliable interconnects [13]. Dynamic reconfiguration time in Immunet is an exponential function of the number of nodes in the system, and for a CMP system of comparable dimensions, can be estimated to be less than 8 000 cycles. This result, however, comes at a very high cost, since each node must include and dynamically update three different routing tables. Because Cardio relies on software routines to reconfigure the hardware, it is slower (up to 50 000 cycles) in responding to failures. But, since fault events are very infrequent, this slowdown has a negligible impact on performance.

F. Performance and Traffic Impact

In this section we study the impact of our solution on fault-free interconnect performance and communication overhead. For this last study we report the extra execution time experienced when running SPECMPI benchmarks and the percentage of extra packets that must be transmitted for diagnostic purposes. We show in Fig. 10(a) that, for most benchmarks, the performance impact is lower than 3% and almost uniform over all topologies. An interesting exception is the *104.milc* benchmark evaluated in the mesh topology, which suffers of a significant performance loss. This is because each core in that benchmark relies on very frequent and long data transfers to one process mapped to the core on the top left corner of the mesh. This core therefore has a much more limited bandwidth,

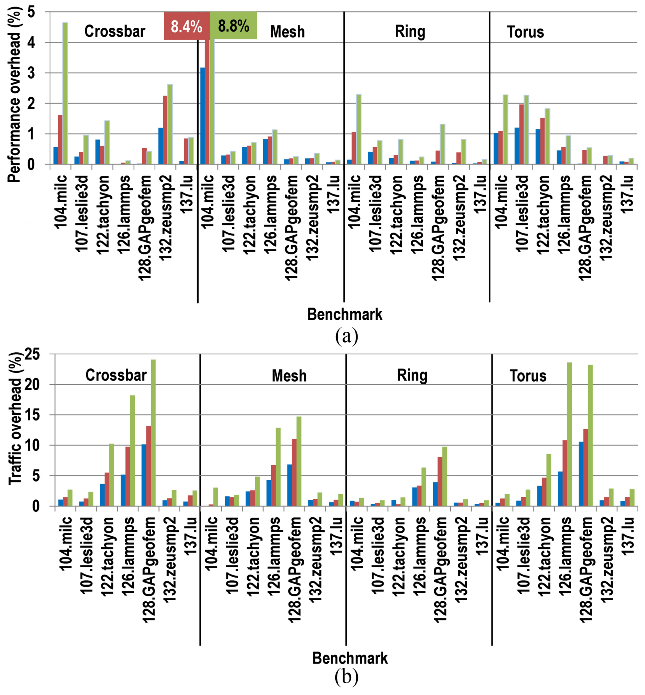

Fig. 10. Performance impact and extra traffic (measured in message*hop) due to interconnect discovery on SPECMPI benchmarks. (a) Performance impact for the considered applications is limited for most benchmarks (3%) and almost uniform over the four topologies. (b) For most applications and topologies the extra communication introduced by Cardio is limited (5% on average).

and thus the performance impact measured in this scenario is particularly pessimistic.

Fig. 10(b) plots the extra traffic introduced by our system. This can also be used as a estimate of the energy overhead due to Cardio. Our design typically introduces less than 10% of extra traffic, and this figure varies greatly with the benchmark considered: its overhead is higher for applications with little intercore communication (e.g., *128.GAPgeofem*).

Other solutions for reliable interconnect, such as Immunet and Vicis, have no performance impact during fault-free operations, but impose much larger area overheads. Immunet requires three different routing tables per node [13], while the overhead for Vicis is more than 40% of the design's baseline area [23]. Furthermore, Cardio provides global knowledge of hardware state to a middleware layer, thus enabling system-level tuning of hardware reconfiguration policies.

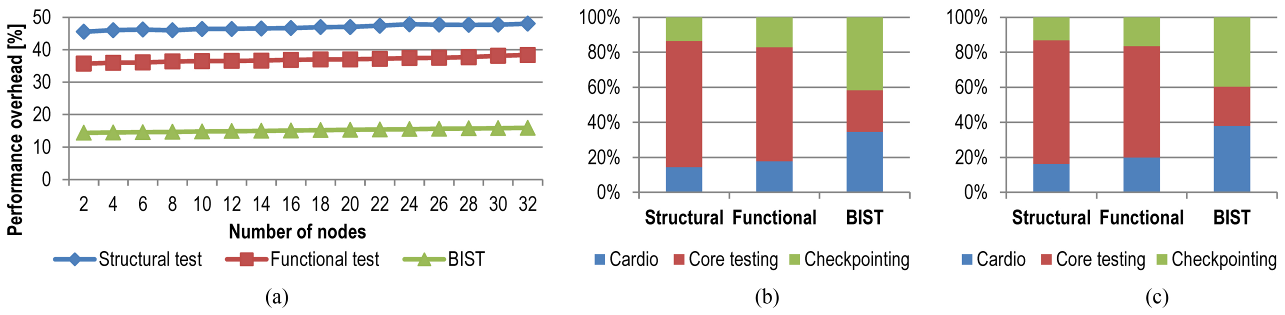

Fig. 11. Analysis of the overhead of the reliability techniques for different CMP configurations. (a) Performance overhead of 16 CMP configuration, including between 2 and 32 cores. We considered three different core testing technique: structural testing [30], functional testing [32], and built-in-self-test [50]. (b) and (c) Distinct performance overhead contributions for configurations containing 16 and 32 cores, respectively.

As the number of nodes in CMPs is expected to increase, it is important to evaluate the scalability of our solution. Assuming a CMP with n cores, the maximum number of messages periodically that must be exchanged in Cardio is $2n^3$ and $2n^2(n-1)$ for a torus and a mesh, respectively. In order to overcome a network failure, the number of messages exchanged in a torus is $8 + 4n^3$ and is $8 + 4n^2(n-1)$ for the mesh. To provide some perspective, a newly discovered fault in a 8x8 torus triggers 1 032 messages in Cardio and 12 240 in Immunet [13]. As the number of processors in CMPs grows, the gap between the numbers of messages exchanged increases even further. For instance, overcoming a failure in a 16x16 torus requires 4 104 and 244 908 messages for Cardio and Immunet, respectively. Finally, reconfiguration time in Immunet increases with the size of the system, while Cardio reacts always within the same time window. This is because Immunet needs to propagate new routing tables from one node to all others, while Cardio detects failures locally and handles hardware reconfigurations in software. Table I compares performance and cost of our hybrid hardware/software solution against a fully hardware approach, Immunet.

G. Full System Performance Analysis

In this section we evaluate Cardio's performance impact on a complete reliable system, which also includes processor checkpointing and testing procedures for all hardware components. A number of previous works developed mechanisms that can be adopted for these purposes, and we rely on them to estimate the cost of deploying a holistic reliable solution.

We use gem5 to evaluate 16 different fault-free CMP configurations executing the MEVBench benchmark suite [48]. In this analysis we consider computational epochs 20 million cycles long, a common choice for such systems. With the goal of analyzing our results in detail, we divide the performance overhead of the system in three different categories: checkpointing, core testing, and Cardio.

Checkpointing—In order to measure the performance impact of a checkpoint system, we assumed that our system deploys ReVive, a complete low-cost checkpointing solution [34]. The goal of this system is to recover the memory state of a shared-memory machine in less than one second. This recovery time is reasonable, since we do not expect hardware failures to manifest frequently enough to impact overall system performance. Hardware checkpoints are taken every 20ms, and consist of logging recently modified data from each node's cache and values stored in the register file. In order to model this system, our experiments account that the average error-free overhead introduced by ReVive is 6.3%, a value extrapolated from previous evaluations of this paer [34]. This figure is rather conservative, since it assumes that the system takes a full checkpoint every 10ms. In our experiments, we account performance penalty due to checkpoint synchronization as overhead for Cardio.

Core testing—Each core independently pauses its execution at the end of an epoch to perform a complete hardware self-test. A number of approaches have been proposed to perform these checks, which vary greatly in fault coverage and cost. In order to measure Cardio's impact with a wide range of hardware self-test solutions, we considered three different techniques: structural testing [30], functional testing [32], and built-in-self-test [50]. Structural and functional test techniques have little impact on the silicon area (up to 6%). . They can achieve high fault coverage, 99.5% and 95%, respectively, for a relatively high performance cost, 3.4 and 5.4 million cycles, respectively. Built-in-self-test techniques achieve a faster test time, only 3 300 cycles, for a higher silicon area cost (up to 15%) and can reach 95% fault coverage. It is worth noting that these evaluations do not consider a reliability-aware operating system, whose scheduling policy can significantly reduce periodic hardware self-test overhead [40].

Cardio—The arrival of any diagnostic message triggers the execution of a simple procedure of the distributed resource manager in each core, which collects information about hardware state and populates its software structures. The end of an epoch is determined individually by each core through a cycle counter, which interrupts all current operations to test processor hardware, checkpoint program state, and broadcast diagnostic information about its hardware. These routines also analyze the graph of the components available in the system, and, if needed, trigger system reconfiguration and notify the OS of possible hardware alterations.

The hardware of the routers in the system is checked as frequently as the cores using tests that require 150 000 cycles [51]. Router's lists of local connections is kept updated much more frequently, every 10 000 cycles, to ensure correct communication between directly connected nodes.

Fig. 11(a) shows the results we obtained for the range of CMP evaluated and the core testing technique considered. These curves show that the total performance overhead slightly worsens as the number of cores increases. This increase in overhead is due to the additional diagnostic messages exchanged by the cores and averages to a performance degradation of roughly 0.05% per additional core. Fig. 11 also reports the single contribution to the total overhead of the three mechanisms we require to ensure reliable computing: checkpointing, hardware testing, and Cardio. Fig. 11(b) and (c) reports these results for a configuration containing 16 cores and 32 cores, respectively. On one hand, systems adopting structural and functional testing suffer a very significant slowdown, which is dominated by the core testing procedures. On the other hand, the configurations employing built-in-self-tests have more balanced performance overheads.

Compared against the baseline system, Cardio alone introduces an overall performance overhead between 4.5% and 7.8%. The minimum performance overhead is reported for the 2-core system deploying built-in-self-test. On the other side of the spectrum, the maximum performance overhead for Cardio is for the 32-core configuration adopting structural testing. Two factors drive the relatively high overhead experienced by this configuration. The first one is the higher number of diagnostic messages that must be handled by Cardio's resource manager due to the higher core count. The second factor is due to the compounded performance penalty of all reliability techniques adopted in this configuration: since performance is hindered by the long self-test routines, Cardio interrupts workload execution more frequently.

H. Area Overhead

We used Cacti 5.3 to estimate the area overhead of our solution [52]. For this paper we assume to deploy Cardio on a system composed of 2-wide out-of-order cores built in 32-nm process technology. Each network interface is enhanced with ten buffers (Section V-B) of 32 bytes each (the size of one packet). In addition, we require ten counters associated with the buffers to track timeouts, and each counter should be 20 bits wide to allow for a wide range of timeout values. Thus, the total additional storage required by each network interface is 345 bytes, which adds to a total of 0.036 mm^2 for a system composed of 16 nodes. Note that this overhead is common to all solutions that require the ability to recover in-flight messages. Each router must store the IDs of the nodes connected to its links and a reconfigurable routing tables, for a total of six bytes. As a comparison, each router in Immunet demands 28 bytes of additional storage. Storage requirements grow linearly with system size, and thus Cardio benefits are even more marked for larger CMPs. When we consider both reconfigurable routing tables and router self-test logic, interconnect area increases by approximately 11.4% [51].

Finally, we estimated that the hardware structures needed for full-system checkpoint add a total of 1.3236 mm^2 to each core, increasing its area by 8.5%. More silicon real-estate may be required for the core self-test mechanism—between 0% and 15%, depending on the deployed technique [30], [32], [50].

VI. Limitations and Future Work

Cardio is a novel solution for dynamically managing unreliable CMP components. In this paper we showed that it can quickly reconfigure systems composed of tens of components to work around hardware failures. Although effective, we recognize that our design is affected by two limitations. First, the periodic broadcast of diagnostic messages introduces a non-negligible performance overhead, as also reported in our experimental evaluation in Section V-F. Second, Cardio relies on the presence of at least one operational general purpose processor to run its resource manager.

In the future, we would like to explore other hardware and software solutions that could lower performance overhead at the cost of, for instance, higher silicon area. We would like to study the possibility of embedding microcontrollers dedicated to executing Cardio's resource manager. Modern microprocessors, such as the Intel Nehalem microprocessor, embed full-fledged microcontrollers with the sole purpose of tuning cores' voltage and frequency. Using similar microcontrollers to manage resource availability and to reconfigure hardware components may help containing the cost of our solution, while also enabling its adoption on heterogeneous SoC designs. We also believe that Cardio could benefit from and ease the development of a reliability- and fault-aware OS.

VII. Conclusion

In this paper we presented Cardio, a novel hardware/software architecture to manage reliability in CMPs. Cardio is a system-level solution based on the periodic exchange of diagnostic messages among hardware components to maintain coherent knowledge of resource availability. We evaluated Cardio on a custom, fault-aware simulator for chip multiprocessors and studied the dynamic capability of Cardio to overcome permanent faults, finding that its reconfiguration time upon fault detection is comprised between 20 and 50 thousand cycles. Finally, we showed that Cardio has a very low impact on overall performance (4.5%) and introduces minimal additional traffic (5%) during normal system operation. Furthermore, this solution is fully distributed and scales well with as the number of cores on a CMP increases.

References

[1] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, B. Liewei, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook,"TILE64 processor: A 64-core SoC with mesh interconnect," in *Proc. Solid-State Circuits Conf.*, 2008, pp. 88–598.

[2] S. Borkar, "Thousand core chips: A technology perspective," in *Proc. Des. Autom. Conf.*, 2007, pp. 746–749.

[3] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Proc. Solid-State Circuits Conf.*, 2010, pp. 108–109.

[4] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," in *Proc. ACM SIGGRAPH*, 2008, pp. 18:1–18:15.

[5] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," in *IEEE Micro.*, vol. 25, no. 6, pp. 10–16, 2005.

[6] A. Strong, E. Wu, R.-P. Vollertsen, J. Sune, G. L. Rosa, T. Sullivan, and R. Stewart, *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. New York, NY, USA: Wiley, 2009.

[7] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *Proc. Conf. Architec. Support Program. Lang. Oper. Syst.*, Mar. 2008, pp. 265–276.

[8] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam, "Sampling + DMR: Practical and low-overhead permanent fault detection," in *Proc. Symp. Comput. Architect.*, 2011, pp. 201–212.
[9] A. Pellegrini, V. Bertacco, and T. Austin, "Fault-based attack of RSA authentication," in *Proc. DATE Conf.*, 2010, pp. 855–860.
[10] A. Pellegrini *et. al*, "CrashTest'ing SWAT: Accurate, gate-level evaluation of symptom-based resiliency solutions," in *Proc. DATE Conf.*, 2012, pp. 1106–1109.
[11] E. Musoll, "Mesh-based many-core performance under process variations: A core yield perspective," *ACM SIGARCH Comput. Architect. News*, vol. 37, no. 4, pp. 27–34, 2010.
[12] S. Shamshiri and K.-T. T. Cheng, "Modeling yield, cost, and quality of a spare-enhanced multicore chip," *IEEE Trans. Comput.*, vol. 60, no. 9, pp. 1246–1259, Sep. 2011.
[13] V. Puente, J. A. Gregorio, F. Vallejo, and R. Beivide, "Immunet: A cheap and robust fault-tolerant packet routing mechanism," in *Proc. Symp. Comput. Architect.*, 2004, pp. 198–209.
[14] P. Zajac, J. Collet, and A. Napieralski, "Self-configuration and reachability metrics in massively defective multiport chips," in *Proc. On-Line Test. Symp.*, 2008, pp. 219–224.
[15] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 87–96, Jan.–Mar. 2004.
[16] D. Bernick *et. al*, "NonStop advanced architecture," in *Proc. Conf. Dependable Syst. Netw.*, 2005, pp. 12–21.
[17] K. Constantinides, S. Shyam, S. Phadke, V. Bertacco, and T. Austin, "Ultra low-cost defect protection for microprocessor pipelines," in *Proc. Conf. Architect. Support Programm. Lang. Oper. Syst.*, 2006, pp. 73–82.
[18] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin, "Tolerating hard faults in microprocessor array structures," in *Proc. Conf. Dependable Syst. Netw.*, 2004, pp. 51–60.
[19] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger, "Exploiting microarchitectural redundancy for defect tolerance," in *Proc. Conf. Comput. Design*, 2003, pp. 481–488.
[20] S. Gupta, S. Feng, A. Ansari, B. Jason, and S. Mahlke, "The StageNet fabric for constructing resilient multicore systems," in *Proc. Symp. Microarchitect.*, 2008, pp. 141–151.
[21] A. Pellegrini, J. Greathouse, and V. Bertacco, "Viper: Virtual pipelines for enhanced reliability," in *Proc. Symp. Comput. Architect.*, 2012, pp. 344–355.
[22] K. Constantinides *et. al*, "BulletProof: A defect-tolerant CMP switch architecture," in *Proc. Symp. High-Performance Comput. Architect.*, 2006, pp. 5–16.
[23] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester, "Vicis: A reliable network for unreliable silicon," in *Proc. Design Automat. Conf.*, 2009, pp. 812–817.
[24] T. Dumitras and R. Marculescu, "On-chip stochastic communication," in *Proc. DATE Conf.*, 2003, pp. 10790–10795.
[25] A. Sanusi and M. Bayoumi, "Smart-flooding: A novel scheme for fault-tolerant NoCs," in *Proc. SOC Conf.*, 2008, pp. 259–262.
[26] T. Bressoud and F. Schneider, "Hypervisor-based fault tolerance," in *Proc. Symp. Oper. Syst. Principles*, 1995, pp. 1–11.
[27] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Adapting to intermittent faults in multicore systems," in *Proc. Conf. Architect. Support Program. Lang. Oper. Syst.*, 2008, pp. 255–264.
[28] J. Blome, S. Feng, S. Gupta, and S. Mahlke, "Self-calibrating online wearout detection," in *Proc. Symp. Microarchitect.*, 2007, pp. 109–122.
[29] P. Singh, C. Zhuo, E. Karl, D. Blaauw, and D. Sylvester, "Sensor-driven reliability and wearout management," *IEEE Design Test Comput.*, vol. 26, no. 6, pp. 40–49, Nov.–Dec. 2009.
[30] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based defect tolerance for chip-multiprocessors," in *Proc. Symp. Microarchitect.*, 2007, pp. 97–108.
[31] S. Gupta, A. Ansari, S. Feng, and S. Mahlke, "Adaptive online testing for efficient hard fault detection," in *Proc. Conf. Comput. Design*, 2009, pp. 343–349.
[32] A. Pellegrini and V. Bertacco, "Application-aware diagnosis of runtime hardware faults," in *Proc. Conf. Computer-Aided Design*, 2010, pp. 487–492.
[33] S. Murali, T. Theocharides, N. Vijaykrishnan, M. Irwin, L. Benini, and G. De Micheli, "Analysis of error recovery schemes for networks on chips," *IEEE Design Test Comput.*, vol. 22, no. 5, pp. 434–442, Sep.-Oct. 2005.
[34] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proc. Symp. Comput. Architect.*, 2002, pp. 111–122.
[35] D. Sorin, M. Martin, M. Hill, and D. Wood, "SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proc. Symp. Comput. Architect.*, 2002, pp. 123–134.
[36] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.
[37] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *Proc. Conf. Fundamentals Comput. Theory*, 1983, pp. 127–140.
[38] D. Fick, A. DeOrio, V. Bertacco, D. Sylvester, and D. Blaauw, "A highly resilient routing algorithm for fault-tolerant NoCs," in *Proc. DATE Conf.*, 2009, pp. 21–26.
[39] Y. Li, M. Samy, and S. Mitra, "CASP: Concurrent autonomous chip self-test using stored test patterns," in *Proc. DATE Conf.*, 2008, pp. 885–890.
[40] Y. Li, O. Mutlu, and S. Mitra, "Operating system scheduling for efficient online self-test in robust systems," in *Proc. Conf. Computer-Aided Design*, Nov. 2009, pp. 201–208.
[41] A. DeOrio, K. Aisopos, V. Bertacco, and L.-S. Peh, "DRAIN: Distributed recovery architecture for inaccessible nodes in multi-core chips," in *Proc. Design Autom. Conf.*, 2011, pp. 912–917.
[42] A. Alaghi, N. Karimi, M. Sedghi, and Z. Navabi, "Online NoC switch fault detection and diagnosis using a high level fault model," in *Proc. IEEE Int. Symp. Defect Fault-Tolerance VLSI Syst.*, 2007, pp. 21–29.
[43] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-down Approach*. Reading, MA, USA: Addison-Wesley, 2009.
[44] D. Bertozzi and L. Benini, "Xpipes: A network-on-chip architecture for gigascale systems-on-chip," *IEEE Circuits Syst. Mag.*, vol. 4, no. 2, pp. 18–31, Sep. 2004.
[45] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, *et. al*, "The gem5 simulator," *SIGARCH Comput. Architect. News*, vol. 39, no. 2, pp. 1–7, 2011.
[46] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, *et. al*, "Autonet: A high-speed, self-configuring local area network using point-to-point links," *IEEE J. Sele. Areas Commun.*, vol. 9, no. 8, pp. 1318–1335, Oct. 1991.
[47] M. S. Muller, K. Kalyanasundaram, G. Gaertner, W. Jones, R. Eigenmann, R. Lieberman, *et. al*, "SPEC HPG benchmarks for high-performance systems," *J. High Perform. Comput. Netw.*, vol. 1, pp. 162–170, 2004.
[48] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "MEVBench: A mobile computer vision benchmarking suite," in *Proc. Symp. Workload Characteriz.*, 2011, pp. 91–102.
[49] I. Loi, F. Angiolini, and L. Benini, "Synthesis of low-overhead configurable source routing tables for network interfaces," in *Proc. DATE Conf.*, 2009, pp. 262–267.
[50] M. Mehrara, M. Attarian, S. Shyam, K. Constantinides, V. Bertacco, and T. Austin, "Low-cost protection against SER upsets and silicon defects," in *Proc. DATE Conf.*, 2007, pp. 1–6.
[51] A. DeOrio, D. Fick, V. Bertacco, D. Sylvester, D. Blaauw, J. Hu, and G. K. Chen, "A reliable routing architecture and algorithm for NoCs," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 31, no. 5, pp. 726–739, May 2012.
[52] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," Hewlett-Packard Laboratories, Palo Alto, CA, USA, Tech. Rep. HPL-2008-20, 2008.

Andrea Pellegrini (S'03–M'13) received the Laurea Magistrale (with honors) in computer engineering from the University of Bologna, Bologna, Italy, in 2007. He is currently pursuing the Ph.D. degree at the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, MI, USA.

His current research interests include microarchitecture, secure computer systems, fault modeling, and fault tolerant designs.

Valeria Bertacco (S'95–M'03–SM'10) received the Laurea degree in computer engineering from the University of Padova, Padua, Italy, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 2003.

She is currently an Associate Professor of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, MI, USA. She was previously associated with Synopsys for four years. Her current research interests include the areas of complete design validation, digital system reliability, and hardware-security assurance.

Dr. Bertacco is the recipient of the IEEE CEDA Early Career Award and has served on the program committees of DAC, DATE, and MICRO.