Cardio: Adaptive CMPs for Reliability through Dynamic Introspective Operation

Andrea Pellegrini and Valeria Bertacco
University of Michigan
{*apellegrini, valeria*}*@umich.edu*

***Abstract*—Current technology scaling enables the integration of tens of processing elements into a single chip, and future technology nodes will soon allow the integration of hundreds of cores per device. While very powerful, many experts agree that these systems will be prone to a significant number of permanent and transient faults during their lifetime. If not properly handled, effects of runtime failures can be dramatic.**

In this work, we propose Cardio, a distributed architecture for reliable chip multiprocessors. Cardio, a novel approach for on-chip reliability is based on hardware detectors that spot failures and on software routines that reorganize the system to work around faulty components. Compared to previous online reliability solutions, Cardio provides failure reactivity comparable to hardware-only reliable solutions while requiring a much lower area overhead. Cardio operates a distributed resource manager to collect health information about components and leverages a robust distributed control mechanism to manage system-level recovery. Our architecture operational as long as at least one general purpose processor is still functional in the chip. We evaluated our design using a custom simulator and estimate its runtime impact on the SPECMPI benchmarks to be lower than 3%. We estimate its dynamic reconfiguration time to be comprised between 20 and 50 thousand cycles per failure.

I. Introduction

Current levels of silicon integration allow designers to fit billions of transistors in a single chip. This steady growth has not yet reached its limit, with future transistor sizes pushing further into the nanometer domain. Thanks to such technological achievements we can now envision future multi-billion transistor chip multiprocessors (CMPs) as extremely complex distributed systems, where processors and memory structures are connected through dedicated interconnect networks [13, 3]. Unfortunately, according to several experts, reliability is a major obstacle that can jeopardize this growth. Indeed, as transistor dimensions reduce, their susceptibility to both transient and permanent faults is expected to increase significantly, causing failures in deployed systems [30].

Chip multiprocessors have been widely adopted in a variety of applications because of their performance advantages and scalability. These architectures are particularly interesting from a reliability standpoint, since single cores can be disabled if faulty [20]. However, most of these systems have very little active support to overcome runtime faults and thus they can face critical failures when defects manifest during operation. Runtime faults can reduce system availability, causing significant financial losses. Even worse, undetected faults can lead to silent data corruptions, potentially compromising system security and causing safety hazards [22]. In this landscape, solutions that dynamically overcome runtime faults are necessary to extend a system's lifespan.

Several recent works have proposed individual mechanisms to detect, diagnose, and recover from faults in microprocessors, memory structures or chip interconnects. Current solutions focus on individual components and therefore do not allow the system as a whole to adapt to runtime failures. Furthermore, most of the reliable architectures proposed in the literature rely only on hardware structures for both fault detection and recovery. Therefore, such solutions incur significant hardware overheads due to the addition of components that are rarely triggered.

Cardio, our proposed solution, is a distributed hardware/software system that manages a CMP's availability at runtime. Our architecture uses hardware detectors to promptly detect hardware faults but delegates all system reconfiguration tasks to software routines. This paradigm for reliable systems allows for a low-cost solution (the only hardware support necessary is for fault detection mechanisms) without compromising its effectiveness in recovering from faults. Cardio distributed hardware manager leverages system-level information collected from all self-testing hardware components, and reconfigures the system to work around faults.

A. Contributions

This work makes the following contributions to the area of online fault recovery and reconfiguration:

- **We introduce a distributed resource manager to handle permanent runtime faults on CMPs.** Hardware components periodically exchange diagnostic messages to report their state. Diagnostic messages are collected at runtime by software routines running on the general purpose cores. Local resource managers use these messages to update knowledge about the system and synchronize to evaluate its health. Cardio does not require any a-priori knowledge of the CMP since it is based on simple message broadcast among its components.
- **We propose a novel routing algorithm that targets networks on chip.** In contrast to other fault-adaptive routing solutions, Cardio relies on hardware mechanisms to detect faults and on software-driven reconfiguration of the underlying hardware interconnect. Our routing algorithm relies on the exchange of diagnostic messages among directly connected network components to build a list of reachable network nodes. Interconnect components then transfer their local knowledge of the directly reachable nodes to a distributed software layer that builds communication routes.

II. Related Work

In this section we overview some previously published works, comparing them against our solution.

CMP reliability through dedicated hardware. A solution for reliability in systems composed of hundreds of cores connected through NoCs has been proposed by Zajac, *et al.* [32]. This design targets systems organized in homogeneous self-testing computation tiles composed of a core and a router. Special hardware units, called input/output port (IOP) are

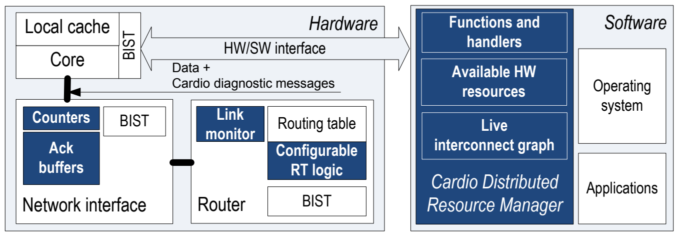

Fig. 1. **Cardio architecture overview.** Cardio hardware and software additions are highlighted in the figure. Communication endpoints are augmented with acknowledgment buffers and counters to determine transmission failures; routers are enhanced with logic to diagnose link-connectivity and to reconfigure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

in charge of discovering and allocating jobs to a system's computational tiles. We recognize three majors drawbacks in this solution: i) it incurs in the extra area necessary to deploy the IOP elements; ii) IOPs are dedicated to manage hardware components and are single point of failures in the system; iii) the proposed architecture does not distinguish between cores and interconnect, thus deactivating an entire computational tile even if only partially defective.

In contrast, our solution relies on the general purpose cores present in the system to manage the hardware and has different the discovery procedures between interconnect and cores, thus avoiding the cost of disabling operational hardware.

Autonomic and organic hardware. Autonomic and organic computing has been envisioned as a solution to create self-organizing and self-managing computer systems [14]. For instance, DodOrg is a project for autonomic robots inspired by biological organisms and organizes the computer system in three layers: cells, organs, and brain [2]. Other works focused on developing network components that can be used in organic systems, with particular focus on communication quality of service [1, 10, 27]. Researchers have studied the possibility of developing autonomic systems on chip, proposing the insertion of autonomic hardware elements to observe and control each SoC functional unit [17]. However, this design adds significant complexity to the system and requires special-purpose hardware to manage the SoC.

Even though Cardio shares a theoretical base with these works, it extends their principles proposing a concrete and viable solution to manage dynamic failures of CMP components.

Reliability in NoCs. In recent years, several researchers have proposed low cost solutions that focus only on reliable intra-chip communication, including network on chip routers capable of detecting and recovering from faults [6]. These solutions usually incur high hardware overheads, proportional to their degree of adaptability to faults and their capability to work on arbitrary topologies. Numerous solutions for adaptive routing algorithms in simple topologies such as meshes and tori are available. However their adoption to other topologies, when possible, is extremely challenging [11].

Stochastic routing and smart-flooding have been proposed as low cost solutions for reliable on-chip communication, but their impact on traffic makes them viable only for lightly loaded networks [9, 25]. Cardio, on the other hand, is agnostic to both topology and routing algorithm. An example of distributed discovery algorithm for reliable NoCs completely developed in hardware is Immunet [24]. This solution can quickly adapt to hardware faults, with a recovery time estimated in less 10,000 cycles for a 8x8 mesh, but burdens the routers to perform the complex algorithms needed to update their local routing tables. Immunet's algorithm dynamically adapts to faults through a chain of updates that involves all nodes in the network. When a fault is detected, the network is flooded with a number of diagnostic messages that grows exponentially with the number of nodes. Cardio, instead, is somewhat slower in reacting to hardware faults in the interconnect, between 20,000 and 50,000 cycles, but relies on much simpler routers and requires the transmission of a limited number of diagnostic messages to recover from a failure.

Since network reconfiguration in Cardio is managed in software, it supports sophisticated routing schemes without increasing the complexity the NoC components. Furthermore, our solution has a system-level knowledge of the CMP, thus even allowing application-aware tuning of packet routes. Neither these capabilities are achievable by the hardware-only NoC reconfiguration algorithms proposed in previous works.

Fault-tolerant microprocessors. Several solutions for online testing and dynamic recovery of microprocessors are available in the literature [7, 8, 12, 21]. However, these works focus on the individual hardware components and lack system-level solutions to the challenges posed by runtime failures in future CMPs. Detecting a failing component is only a first step towards ensuring that the whole CMP can still be functional and these works are thus orthogonal to our solution.

Reliability via middleware. The idea to adopt a middleware layer to support hardware reliability was first investigated by Bressoud, *et al.* [5]. Their work provides a high cost (100% performance overhead) reliability solution through software execution replication. More recently, middleware-based reliability solutions were proposed to tackle intermittent faults [31]. Cardio, on the other hand, utilizes hardware fault detectors to quickly recognize permanent failures and sporadically triggers the execution of software routines to react to them.

III. Cardio Architecture

Cardio follows an event notification-reaction paradigm well-suited to the reliability needs of future CMPs. Two problems need to be addressed to ensure that a CMP system subjected to hardware failures can remain operational. First, hardware components in the system need to be tested and categorized as either functional or unavailable. The second issue is to determine how healthy components are connected and how to ensure communication among them. Typical on-chip reliability solutions rely on hardware structures to achieve both these goals. In contrast, Cardio is based on the capability of hardware components in the CMP to self-test their functionalities and share such information with the rest of the system. The hardware fault detectors interact with a lightweight software layer, the resource manager. Each general purpose core runs an instance of the resource manager, and maintains and organizes information about the on-chip hardware components. Figure 1 shows a high level schematic of the additional components required for a Cardio-enabled CMP design.

During normal operation, application execution is divided in epochs whose length is established by the period of the tests executed on the hardware. Current executions are always considered speculative. Output and state of an epoch are committed to a safe checkpoint only when all the hardware components that contributed to the outcome of the application are detected as healthy. In-flight communications are temporarily stored in buffers to allow packet retransmission.

While the application is speculatively executing an epoch, local hardware components, such as processors and NoC routers, periodically and independently pause their tasks to test the integrity of their hardware. Hardware tests on NoC components are not limited to their internal logic but are also performed to their local connections. These local tests are not globally synchronized, and at least one hardware self-test needs to be performed within the duration of an epoch.

The outcome of the local hardware tests is then sent to the rest of the system. Not all local tests require a full-system result broadcast: for instance, if two neighboring NoC routers do not see any alteration in the status of the link connecting them, there is no need to update the resource managers. More frequent diagnostic tests lead to more reactive systems but also cause higher performance impact and diagnostic message proliferation. Their frequency is thus a design trade-off between extra traffic experienced in the system and reactivity to faults (analyzed in Section V).

Diagnostic messages broadcasted to the system are collected by the local distributed managers and update two software structures that keep hardware state: a list of available hardware resources and a live interconnect graph. The first lists all hardware resources currently available in the CMP. The second structure is the live interconnect graph, and is used to map the active links and NoC components in the design. This latter structure is used to compute the routes that NoC packets will follow. Local managers synchronize among each other to make sure they have a coherent vision of the system. If no failures are detected, the local managers allow the checkpoint system to commit the previously executed epoch. Otherwise, if the local managers detect changes in the hardware of the CMP, current speculative executions are discarded and the operating system will remap the application on the available resources.

Note that, after a hardware failure, the state of each active component comprising the CMP system needs to be recovered to restart software execution. For this purpose, Cardio can rely on either software or hardware full-system checkpoint techniques [23, 29]. The notification-reaction paradigm developed in Cardio can also be used to maintain an accurate state of network traffic, local components usage, and temperature. As we show in this work, dynamic reaction mechanisms based on diagnostic message broadcast can be very responsive without severely hinder system's performance.

Dealing with errors occurring in the self-testing logic is an extremely complex problem, that all solutions that perform online hardware tests need to cope with. In this work we do not address this issue, and assume that faults in the self-test logic will cause the whole component to be non testable and thus conservatively mark it as faulty.

IV. Cardio Operation

Cardio is based on periodic exchanges of diagnostic messages among the CMP's hardware modules. When a failure occurs, the distributed resource manager is notified of the problem and software routines to reconfigure the hardware are triggered. The problem of reaching a common decision among several components is an instance of the Byzantine generals' problem [16]. Solving such a problem in this case consists of providing a common knowledge of the system's healthy resources among all the instances of the distributed resource manager. Cardio provides an efficient solution to this problem for CMPs. Hardware discovery and reconfiguration proceed differently depending on whether a core or router failed.

A. Core Monitoring

To gather and distribute system-level knowledge about the health of the cores in a CMP, the processors follow the sequence of operations illustrated in Figure 2.

Each core independently and periodically suspends its normal execution to perform self-tests (step 2 in Figure 2). Several techniques have been proposed for checking the health of microprocessors at runtime, ranging from structural to functional testing [7, 21]. If the self-test is successful, a checkpoint of the local state of the hardware is taken, as shown in step 3 in Figure 2. Test outcome is then enveloped in a diagnostic message that is broadcasted to the whole system - step 4 in Figure 2. Each diagnostic message is marked with an identifier that is unique to the core that generated it. Depending on the granularity of the test performed on the cores, more detailed information about faulty cores' capability can be provided. For instance, a core whose floating point unit is not functional might still be reported as alive, but only capable of executing integer instructions. Since diagnostic messages are broadcasted throughout the system, every node will eventually receive at least one diagnostic message from another core in the same connected region of the chip.

Still, local checkpoints are required to be synchronized to allow for the system to be recoverable to a coherent state in case some other core failed. In this work, differently from SafetyNet [29], checkpoints are coordinated across the system periodically. Local checkpoints are committed only when diagnostic information generated from all general purpose cores that contributed to application's execution are received by each core's local resource manager. Note that, while waiting for the diagnostic messages, the cores can speculatively proceed with their computations. For instance, in step 5 of Figure 2, core 0 is waiting for the health message from core 3 to commit its local checkpoint. If n is the number of healthy cores in the CMP, each core can receive a maximum of $n - 1$ unique diagnostic messages from other cores. Only when all of the diagnostic messages from the cores that contributed to the application are received a core can safely commit its checkpoint, as shown in step 6 of Figure 2. Otherwise, a message is sent to all cores in the CMP to discard their speculative execution and roll back to a previous safe checkpoint.

To update the knowledge of the healthy hardware, each resource manager builds a list of available cores from the diagnostic messages received. Moreover, the diagnostic messages require a synchronization mechanism to avoid any core from receiving them after the deadline imposed by the periodic hardware test. Since self-test and diagnostic message generation are handled in software, such events can be synchronized through real-time counters. The introspective operations performed by Cardio rely heavily on such timers, and their hardware should be tested thoroughly and frequently or even duplicated to ensure their functionality. In order to avoid problems related to the skew of having multiple real time counters in the system, we broadcast diagnostic messages with a period which is half of the checkpoint interval.

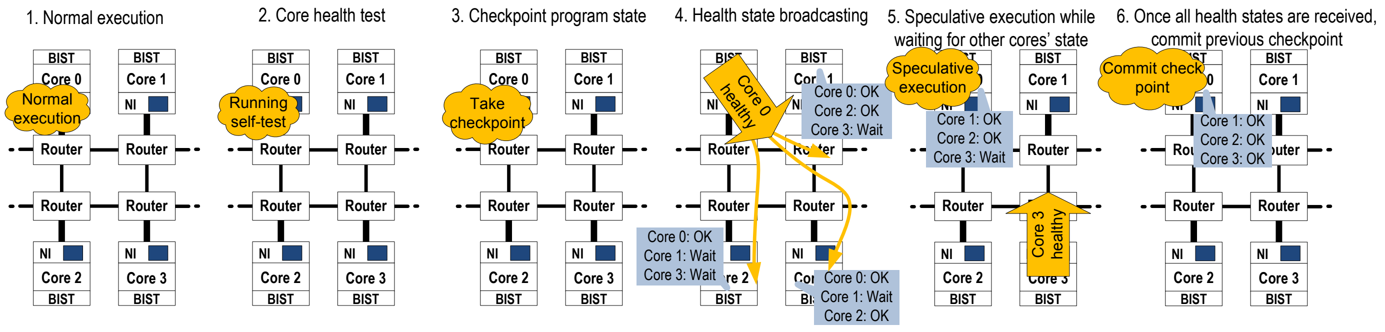

Fig. 2. **Core monitoring and recovery in Cardio.** To maintain an updated state of the available cores in the system, Cardio relies on a five step sequence. 1) The cores perform their normal functions. 2) Core 0, independently from any other core, executes a self-test procedure to detect potential permanent failures. 3) If the test completes successfully, a local checkpoint of the current CPU state is taken. 4) A diagnostic message is broadcasted to the other cores to signal that core 0 is functional. 5) Before core 0 can commit its checkpoint, it needs to receive successful health acknowledgments from all cores that were functional in the previous execution period. Still, it can speculatively continue its execution. 6) Finally core 0 receives the last positive health acknowledgment from core 3 and commits its previous checkpoint.

```
 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 consists of a health check of the hardware of the component. 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.

To guarantee that all diagnostic messages will meet the deadline imposed by the resource manager updates, diagnostic messages' arrival needs to precede such a deadline by at least the longest time needed for any message to reach all cores in the system. Updates for healthy hardware components need to be synchronized among all cores based on the arrival time of the last diagnostic message. Note that a faulty core is not required to advertise its status to the rest of the system: in this case, the distributed hardware manager will report it as unavailable and the other local resource managers will detect the failure at the beginning of their next self-test period.

B. *Interconnect Monitoring*

Correctness and performance of the interconnect is vital for any CMP. In-chip routers are in charge of delivering messages between cores and multiple cores can be connected to a single router. Cardio can be successfully adopted in any NoC topology, and the current health state of the NoC is maintained by the Cardio's local resource managers. We propose a routing algorithm that dynamically discovers network topology and updates communication routes. Two families of routing algorithms are available to dynamically discover and configure an arbitrary network: link-state and distance-vector [15]. For our on-chip dynamic discovery system, we decided to adopt a routing algorithm inspired by link-state. These protocols converge quicker and are more scalable than those based on distance vector, even though they rely on more complex algorithm and have larger memory footprints.

Figure 3 shows the built-in-self-test steps performed by each router when the online testing procedure is activated. As a first step, each network router independently pauses and performs a self-test of its own hardware (step 2 in Figure 4), perhaps using online NoC testing techniques proposed in the literature [6, 11]. The outcome of this first test determines if the hardware can safely operate on each input and output link. The next step performed by the router is the dynamic discovery of the nodes connect at the end of each link. Local link discovery is performed in hardware, independently for each node in the NoC. The link monitors, local to each interconnect node, are dynamically updated through a distributed discovery routine: periodically they generate a discovery "heart beat" that is forwarded through the interconnect to all adjacent nodes to check link integrity, as illustrated in step 3 of Figure 4. The "heart beat" consists of a discovery request message to which the recipient responds with a discovery response and its node ID (step 4 in Figure). Collecting these responses, link monitors populate a table where each local link is associated with the ID of the node directly connected to it (step 5 in Figure). Note that each NoC endpoint needs an extra entry for the unique identifier of the core directly connected to it.

If, after a user-configured amount of time, a link monitor detects that its list of directly connected nodes is missing an entry (perhaps because communication through one link is obstructed by a fault), all packets that were directed towards that link are discarded and the updated local link table is broadcasted to the rest of the system (step 6 in Figure 4). Due to storage and performance constraints, the interval between interconnect checks should be limited to few thousand cycles as discussed in Section 5.3. Advertisements about changes in a local link table signal to the general purpose cores that the network topology was altered. Since congested links can be detected as broken, the discovery procedure is always run on all links to detect those that were previously congested. This also allows parts of the network that were previously temporarily not operational (for example due to intermittent faults) to later be marked as available.

Since network failures might cause some packets in flight to be discarded, a retry mechanism is necessary to avoid communication loss. Cardio addresses such transmission glitches through an acknowledgment protocol: every time a message successfully reaches its destination, the receiver notifies the sender. All interconnect endpoints maintain hardware counters and buffer any pending communications waiting for an acknowledgment. The counters are incremented every cycle and trigger a time-out signal if acknowledgment messages are not

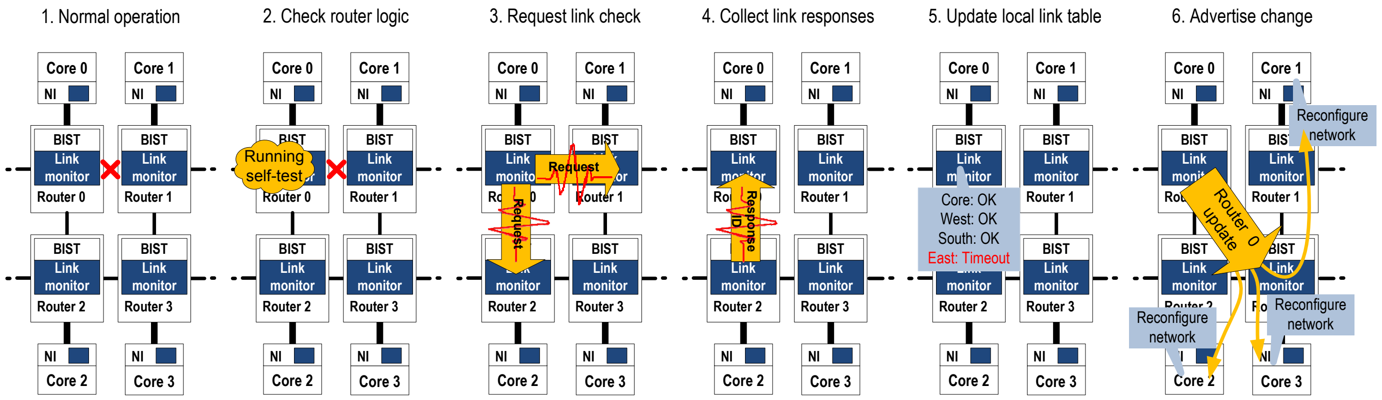

Fig. 4. **Dynamic interconnect management in Cardio.** Self-discovery and reconfiguration in the interconnect are organized in five steps. To reduce the amount of extra traffic in the CMP, only topology *changes* are advertised. In the figure: 1) The NoC performs its normal functions. 2) Router 0 pauses its execution to perform a self-test routine. 3) Since its hardware is still functional, discovery messages are sent to all output links. 4) Router 2 replies to the request with its router ID. 5) No response is received from router 1 within the deadline imposed by the timeout, thus detecting the failed link. 6) Because of this topology change Router 0 broadcasts a diagnostic update that is received by all connected resource managers to reconfigure the network.

received within a certain time threshold. In case of time-out, the network interface will retransmit the failed message; if the second attempt is also unsuccessful, the cores are notified of the problem via an exception. Acknowledgments may be sent through specialized packets or may be piggybacked to regular data packets. Acknowledgment buffers size is a storage and performance trade-off that is evaluated in Section V.

C. Cardio Distributed Resource Manager

In Cardio, the distributed resource manager is in charge of monitoring and managing the system's reconfigurable hardware. With the information collected from the local hardware tests, a light-weight software layer can estimate which resources are available in the device and how to access them.

Resource managers use the information about local connections broadcasted by the link monitors to create a graph of the entire interconnect. All cores in a connected region will reconstruct the same topology: if the interconnect is partitioned into multiple disconnected regions, each core running an instance of the Cardio local resource manager will reconstruct the topology of the region to which it belongs. Once built the interconnect graph, the local resource manager computes all routing paths, thus configuring the system to allow communication among the available hardware. Then, from the diagnostic messages sent by the individual cores, each resource manager reconstructs the list of operational processors. Each local resource manager generates the list of the available hardware and the interconnect graph with the same algorithm. A checksum of the two data structures can be transmitted to all cores in the design to verify that each instance of the resource manager agrees on the current state of the system. If two resource managers disagree on the available hardware components, a further negotiation among the local resource managers is initiated.

The resource manager shares processor resources with other applications running in the system: each different instance of the resource manager periodically interrupts the core's functionality to allow the execution of hardware tests on the core. If no fault is detected in the underlying hardware, the hardware state is checkpointed and the manager advertises test success to all the other cores. The manager then allows computations in the core to speculatively continue while it monitors for notifications about hardware test success from other healthy cores in the system. A next checkpoint is not taken unless advertisements from all previously healthy cores have been received.

Note that cores advertise their state to the rest of the system when the self-tests succeed. Based on the fault coverage achieved by the online testing mechanism deployed for each component, there might be a possibility that a fault could cause a failing component to incorrectly advertise its state as healthy. The probability of such events can be arbitrarily reduced based on the quality of the periodic tests applied to the hardware. If a core does not successfully report as healthy at the time a new system-level checkpoint needs to be taken, a special message is sent to all cores in the system to rollback to the previous synchronized checkpoint, thus preventing processors from committing speculative results that could have been affected by the faulty core. A reliability-aware operating system will then migrate the running application to map the available resources to avoid the usage of the faulty component. Note that if a region of the system becomes isolated due to a failure in the interconnect, the checkpointed state of the cores in that region cannot be retrieved. Solutions to recover the memory content of isolated nodes are beyond the scope of this paper and will be the focus of future research.

V. Experimental Results

We developed a distributed mechanism to manage and organize on-chip resources at runtime. Our solution adds extra traffic due to the diagnostic messages exchanged by the self-checking hardware components. Thus our experiments focus on measuring Cardio's impact on the system interconnect, considering a variety of topologies and workloads. We first focus on finding the optimal size of the acknowledgment buffers at the NoC endpoints and the ideal interconnect discovery period for the nodes constituting the interconnect. We then evaluate the capability of our solution to dynamically overcome failures. Finally, we measure its performance impact on several applications.

In order to examine how Cardio reacts to failures, we measure how communication latency is affected by the occurrence and presence of hardware faults. Finally, we report the impact of our solution on interconnect performance and energy, measuring extra traffic.

A. *Experimental Setup*

We perform our experiments using a fault-aware system-level C++-based simulator working at the transaction-level model, where communication details are separated from the implementation details of functional units. Functional units are modeled through clock counters, and the interconnect implements a cycle-accurate packet-switching system at the packet granularity (we do not consider flit-level structures). Two fault models have been developed on the interconnect links: all packets attempting to traverse a broken link are dropped *(drop-packets)*, or the communication path is blocked at a certain link *(hold-packets)*.

The CMP simulated in our experiments consists of 16 cores, each connected to a dedicated network interface. We considered four different interconnect topologies: ring, mesh, torus and crossbar. The system frequency is set at 2.4GHz, with five-stage routers transferring packets 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* algorithm [26]. Cardio does not impose limitations on the routing algorithm adopted and design choices in this work were made in order to facilitate simulation troubleshooting.

Both uniform random traffic and traces from the SPECMPI benchmark suite [19] are used as input stimuli. 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 offer a more realistic model to evaluate Cardio's performance and traffic overhead. For the random benchmarks we report the packet injection rate as the probability that a core can inject a new packet in the network (in percentage). For the latter benchmarks, SPECMPI applications were instrumented through the Tuning and Analysis Utilities to obtain traces of the communication patterns among the cores executing the applications [28]. To contain simulation time, we reduced the number of cycles between MPI transactions, thus the performance overhead we report for these benchmarks is worse than in their original form.

B. *Acknowledgment Buffer Sizing*

Cardio considers all in flight point-to-point communication to be speculative until the sender receives a confirmation from the receiver. Thus, each transmitted packet that is not broadcasted 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 due to the insertion of packet acknowledgment buffers in the network interfaces. We evaluate several buffer sizes, ranging from 1 data packet (that is, the network interface must receive acknowledgment for one packet before transmitting the next), up to 100 outstanding packets. No faults were injected for this experiment. Figure 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 input a new message

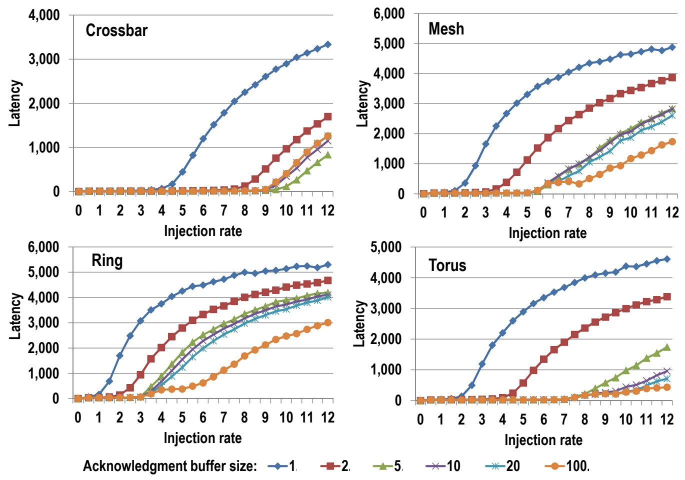

Fig. 5. **Packet latency vs. injection rate for different acknowledgment buffer sizes.** Each curve represents a different acknowledgment buffer size as indicated in the legend. The X-axes represent the probability for each node (in percentage) to inject a new packet in the network. Packet buffers of size 10 provide the best trade-off between storage requirements and packet latency for all analyzed topologies. Injection rate is measured in probability (percentage) of packet injection per core.

in the interconnect at any given clock cycle, while packet latency is measured as the number of cycles from when a data packet is transmitted to when it is received by the destination. We found that, for all topologies, an acknowledgment buffer of 10 packets is a reasonable compromise between storage requirements and packet latency. Indeed, acknowledgment buffers of less than 10 data packets significantly hinder the average packet latency, while even doubling their size provides minimal benefits. Thus, in all subsequent experiments we set the acknowledgment buffer to store 10 outstanding packets. Interestingly, the latency curves we observed level off after the traffic injection rate passes a traffic threshold dependent on the size of the buffers. This behavior is due to limitations in the injected traffic due to the size of the acknowledgment buffers, and follows the trends showed in [4].

C. *Dynamic Discovery Period*

We then analyzed the amount of overhead imposed by the discovery packets exchanged in the interconnect. To do so, we varied the interconnect discovery period from 1,000 to 20,000 cycles. No faults were enabled in this experiment and the interconnect was subjected to a moderate load. We selected a traffic injection rate of 5% because, from the data gathered in our analyses, we observed that resource contention in the network starts to impact packet latency at higher rates. As expected, as the period between interconnect discoveries increases, average packet latency decreases because of bandwidth limitations. As shown in Figure 6, the trend is steeper for topologies such as mesh and ring, for which links are subjected to a higher baseline latency and contention. Given the results obtained in this test, 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. *Dynamic Fault Behavior*

In this section we study the dynamic behavior of Cardio on a mesh at the time of occurrence of a permanent fault, evaluating its reactivity in detecting and overcoming the failure. For this experiment, a system with no faults executes for 150,000

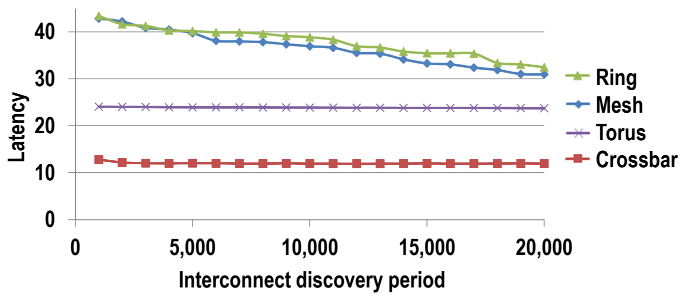

Fig. 6. **Packet latency vs. discovery period.** The impact of the discovery period differs for different topologies: mesh and ring are more sensitive to variations due to a smaller network bisection.

cycles to reach a steady state, and then a link is randomly selected and marked as faulty. Two link fault models were considered for this study: the first causes the broken link to drop all packets traversing it *(drop-packets)*, while the second causes packets to be held at the link, clogging the network *(hold-packets)*. To provide insights on the dynamic behavior of Cardio, we analyze the system at 500-cycle intervals and report, on the Y-axis, *the average latency incurred by all packets* generated during each analyzed window. In this experiment we consider 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 found that the time required for the distributed resource manager to recompute the routing tables is approximately constant at 10,000 cycles. We also computed 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 [18]. We began the network discovery period at the time of fault injection to demonstrate the worst-case performance of our solution. In our evaluation, we disregard the extra traffic introduced by core diagnostic messages, since their transmission frequency is three orders of magnitude lower than for the interconnect components [6].

Results from the *drop-packet* fault model are reported in Figure 7.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 re-transmitted due to the faulty link. After a certain amount of time, directly related to the network discovery period, the routers detect the problem locally and advertise the change in the system's state. The first interconnect reconfiguration process causes the network to temporarily stall, resulting in the second peak observable in the graph. The third peak observable in the graph is caused by a second system reconfiguration, triggered by nodes that may detect the faulty link at different times.

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 shown by the much higher and longer average latency experienced. Indeed, the fault causes congestion among several nodes: the buffers at the nodes connected through the broken link fill up and cause a domino effect to their neighbors and to the rest of the network. As reported in Figure 7.b), the longer the system takes to detect and address the fault, the more dramatic the fault's effects on the overall system.

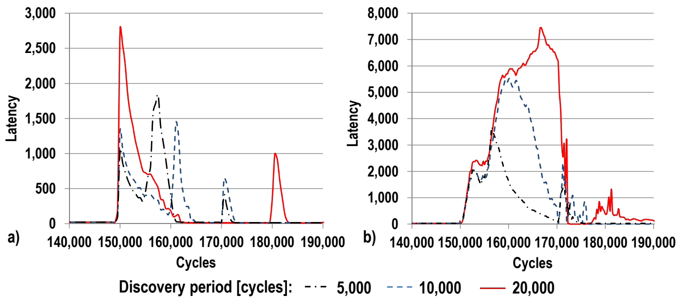

Fig. 7. **Effect of a runtime fault on a link.** This graph plots the average time necessary for a packet to reach its destination; packet latency is averaged between all packets generated in a window of 500 cycles. In this scenarios the link is broken at cycle 150,000 and two fault models are considered: **a)** *drop-packet*; **b)** *hold-packet*.

E. Performance and Traffic Impact

We then study the impact of our solution on 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. During this experiment all topologies are fault-free. We show in Figure 8.a) that, for most benchmarks, the performance impact, and even for very frequent discovery intervals, 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 with a single thread, which is mapped to the core on the top left corner of the mesh - and thus has limited bandwidth compared to the central nodes. The performance impact measured in this scenario is then particularly pessimistic.

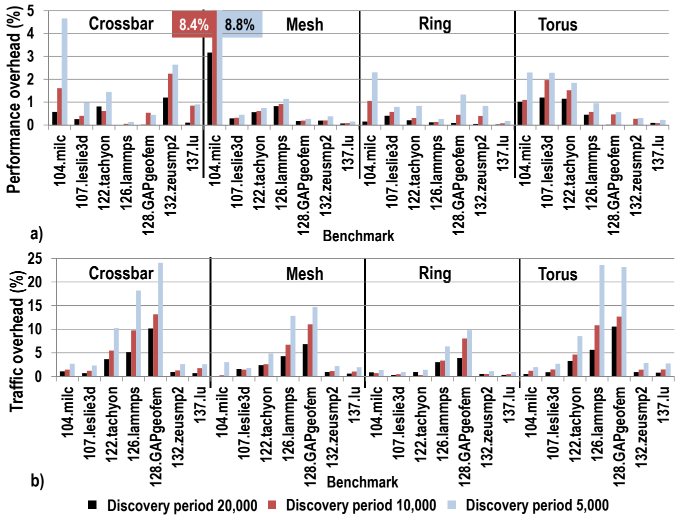

Fig. 8. **Performance impact and extra traffic (measured in message*hop) due to interconnect discovery on SPECMPI benchmarks.** a) The performance impact for the considered applications is limited for most benchmarks (3%) and almost uniform over the 4 topologies. b) For most applications and topologies the extra communication is very limited (5% on average), and both application behavior and topology impact the traffic overhead.

Figure 8.b) shows the percentage of extra traffic introduced by our system. This graph also shows a rough estimate of the energy overhead caused by Cardio in terms of number of extra messages transmitted and received for each benchmark. For most benchmarks and topologies, the number of extra packets due to Cardio's diagnostic messages is less than 10%, and it varies greatly based on the benchmark considered. The impact of discovery messages in the system is higher for applications with little inter-core communication (e.g. *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 [24], while the overhead for Vicis is more than 40% of the design's baseline area [11]. Furthermore, Cardio provides global knowledge of hardware state to a middleware layer, thus enabling tuning of hardware reconfiguration policies at the system level.

F. Area Overhead

Compared to a typical CMP system with precomputed routing tables, Cardio requires the addition of a few hardware components throughout the interconnect. Considering the baseline design used in our evaluation, each network interface must to be enhanced with 10 buffers (Section 5.2) of 32 bytes each (size of 1 packet). In addition, we require 10 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 storage overhead for each network interface is 345 bytes. Note that this latter overhead is common to all solution that need to recover messages that might be in flight when a fault occurs. The total storage requirements at each router is 6 bytes. For comparison, each router in Immunet demands 28 bytes of additional storage. With larger interconnects both these trends grow linearly, and thus Cardio benefits are even more marked.

VI. Conclusions

In this work we presented Cardio, a novel architecture hardware/software architecture to manage reliability in complex CMP systems. Cardio is a system-level solution based on periodic exchanges of diagnostic messages among system's components to maintain coherent knowledge of hardware health among all its components. We evaluated Cardio on a custom, fault-aware simulator for chip multiprocessors and studied the dynamic capability of Cardio to overcome permanent faults, showing that its reconfiguration time is comprised between 20,000 and 50,000 cycles. Finally, we showed that Cardio has a very low impact on performance (3%) and introduces minimal additional traffic (5%) during normal system operation.

References

[1] M. Al Faruque, T. Ebi, and J. Henkel. Configurable links for runtime adaptive on-chip communication. In *Proc. of the Design, Automation and Test in Europe Conference*, Apr 2009.

[2] J. Becker et al. Digital on-demand computing organism for real-time systems. In *Intl. Conference on Architecture of Computing Systems*, Mar 2006.

[3] S. Bell et al. Tile64 processor: A 64-core soc with mesh interconnect. In *Proc. of the Solid-State Circuits Conference*, Feb 2008.

[4] P. Bogdan and R. Marculescu. Quantum-like effects in network-on-chip buffers behavior. In *Proc. of the Design Automation Conference*, Jun 2007.

[5] T. Bressoud and F. Schneider. Hypervisor-based fault tolerance. In *Proc. of the Symposium on Operating Systems Principles*, Dec 1995.

[6] K. Constantinides, J. Blome, S. Plaza, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. BulletProof: a defect-tolerant CMP switch architecture. In *Proc. of the Intl. Symposium on High-Performance Computer Architecture*, Feb. 2006.

[7] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based defect tolerance for chip-multiprocessors. In *Proc. of the Intl. Symposium on Microarchitecture*, Dec. 2007.

[8] K. Constantinides, S. Shyam, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *Proc. of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[9] T. Dumitras and R. Marculescu. On-chip stochastic communication. In *Proc. of the Design, Automation and Test in Europe Conference*, Mar 2003.

[10] T. Ebi, M. Al Faruque, and J. Henkel. Neuronoc: neural network inspired runtime adaptation for an on-chip communication architecture. In *Proc. of the conference on hardware/software codesign and system synthesis*, Oct 2010.

[11] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester. Vicis: a reliable network for unreliable silicon. In *Proc. of the Design Automation Conference*, Jul 2009.

[12] S. Gupta, S. Feng, A. Ansari, B. Jason, and S. Mahlke. The stagenet fabric for constructing resilient multicore systems. In *Proc. of the Intl. Symposium on Microarchitecture*, Dec 2008.

[13] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proc. of the Solid-State Circuits Conference*, Feb 2010.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, Jan 2003.

[15] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley Publishing Company, USA, 5th edition, 2009.

[16] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, Jul 1982.

[17] G. Lipsa and A. Herkersdorf. Towards a framework and a design methodology for autonomic SoC. In *Proc. of the Intl. Conference on Autonomic Computing*, Jun 2005.

[18] I. Loi, F. Angiolini, and L. Benini. Synthesis of low-overhead configurable source routing tables for network interfaces. In *Proc. of the Design, Automation and Test in Europe Conference*, Apr 2009.

[19] M. S. Muller, K. Kalyanasundaram, G. Gaertner, W. Jones, R. Eigenmann, R. Lieberman, M. V. Waveren, and B. Whitney. SPEC HPG benchmarks for high-performance systems. *Intl. Journal of High Performance Computing and Networking*, Jan 2004.

[20] E. Musoll. Mesh-based many-core performance under process variations: a core yield perspective. *ACM SIGARCH Computer Architecture News*, Sep 2009.

[21] A. Pellegrini and V. Bertacco. Application-Aware diagnosis of runtime hardware faults. In *Proc. of the Intl. Conference on Computer-Aided Design*, Oct 2010.

[22] A. Pellegrini, V. Bertacco, and T. Austin. Fault-based attack of RSA authentication. In *Proc. of the Design, Automation and Test in Europe Conference*, Mar 2010.

[23] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proc. of the Intl. Symposium on Computer Architecture*, May 2002.

[24] V. Puente, J. A. Gregorio, F. Vallejo, and R. Beivide. Immunet: A cheap and robust fault-tolerant packet routing mechanism. *ACM SIGARCH Computer Architecture News*, Mar 2004.

[25] A. Sanusi and M. Bayoumi. Smart-flooding: A novel scheme for fault-tolerant nocs. In *Intl. SOC Conference*, Sep 2008.

[26] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, Oct 1991.

[27] C. Schuck, S. Lamparth, and J. Becker. artNoC - a novel multi-functional router architecture for organic computing. In *Intl. Conference on Field Programmable Logic and Applications*, Aug 2007.

[28] S. Shende and A. Malony. The Tau parallel performance system. *Intl. Journal of High Performance Computing Applications*, May 2006.

[29] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. of the Intl. Symposium on Computer Architecture*, May 2002.

[30] A. Strong et al. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. Wiley Press, 2009.

[31] P. M. Wells, K. Chakraborty, and G. S. Sohi. Adapting to intermittent faults in multicore systems. In *Proc. of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Mar 2008.

[32] P. Zajac, J. Collet, and A. Napieralski. Self-configuration and reachability metrics in massively defective multiport chips. In *Proc. of the Intl. On-Line Testing Symposium*, Jul 2008.