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 are soon expected to reach 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 the life of the chip. If not properly handled, the effects of runtime failures can be dramatic.

In this work, we propose Cardio, a distributed architecture for reliable chip multiprocessors. Cardio 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 can provide 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. Cardio can offer recovery so 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%. With a dynamic reconfiguration time between 20 and 50 thousand cycles per failure, the distributed resource manager can detect and setup the system to reconfigure around broken processor cores or interconnect elements.

1. 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. Such technological achievements will allow next generation digital systems to integrate even more transistors. We envision future multi-billion transistor chip multiprocessors (CMPs) as extremely complex distributed systems, where processors and memory structures are connected through dedicated interconnect networks [11]. Unfortunately, according to several experts, reliability is a major factor that can jeopardize this growth. Indeed, as transistor sizes reduce, their susceptibility to both transient and permanent faults is expected to increase significantly, causing failures in deployed systems [28].

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 found to be faulty. Manufacturers exploit these architectures to maintain high production yields in the face of fabrication defects, for instance disabling cores that are not fully operational during post-production testing [19]. However, most of this systems do not have active mechanisms to overcome runtime faults. Thus, they can face critical failures when defects manifest during operation. Runtime faults can reduce system availability, causing significant financial losses. Furthermore, undetected faults can lead to silent data corruptions, potentially compromising system security and causing safety hazards [21]. 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 to manage a CMP's availability at runtime. Our proposed solution 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 (only fault detection mechanisms are required in hardware) without compromising its effectiveness in recovering from faults. The Cardio distributed hardware manager leverages system-level information collected from all self-testing hardware components, and makes global decisions about reconfiguring the system to work around faults. A CMP architecture equipped with Cardio can also improve its mean time to repair and mean down time, thus reducing total system maintenance costs.

1.1 Contributions

We propose a distributed, system-level solution to manage faulty components in CMPs. The performance impact of Cardio is minimal, lower than 3% on average during normal operation. Thanks to its software support, it requires very few hardware changes to a baseline system. 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 and 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.

In this work we target CMPs consisting of up to hundreds of processors, where communication among components is through either explicit message passing or shared memory.

2. RELATED WORK

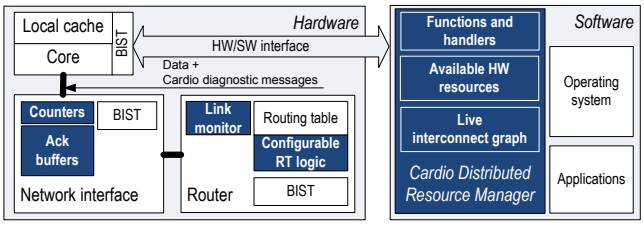

Figure 1: Cardio architecture overview. Cardio hardware and software additions are highlighted in the figure. Communication endpoints are augmented with acknowledgment buffers and counters to determine transmission failures; routers are enhanced with logic to diagnose link-connectivity and to reconfigure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

In 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.* [29]. This design targets systems organized in homogeneous self-testing computation tiles composed of a core and a router. Special hardware cores, called input/output port (IOP) are in charge of discovering and allocating jobs to 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 causing a whole computational tile to become inactive even if it is only partially defective.

In contrast, our solution relies on the general purpose cores present in the system to manage the hardware and distinguishes the discovery procedure between interconnect and core, 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 [12]. For instance, DodOrg is a project for autonomic robots inspired by biological organisms, organizing the system in three levels: 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, 8, 25]. 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 [16]. 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 [4]. These solutions usually incur in 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 [9].

Stochastic routing has been proposed as a low cost solution for reliable on-chip communication, but its impact on traffic makes such solution viable only for lightly loaded networks [7]. Cardio, on the other hand, is agnostic to the topology and routing algorithm. An example of distributed discovery algorithm for reliable NoCs completely developed in hardware is Immunet [23]. This solution can quickly adapt to hardware faults, with a recovery time estimated in less 10,000 cycles for a 8x8 mesh, but demands to the routers to perform complex algorithms 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, it floods the network 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 can support sophisticated routing schemes without burdening NoC components with the extra logic needed to support such algorithms. Furthermore, our solution has a system-level knowledge of the current state 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 [5, 6, 10, 15, 20]. 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 be still 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.* [3]. Their work provides a high cost (100% performance overhead) reliability solution through software execution replication. Cardio, on the other hand, utilizes hardware fault detectors to quickly recognize a failure and sporadically triggers the execution of software routines to react to hardware failures.

3. 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, enough functional hardware resources must be present to achieve a determined task. Given that enough hardware components are available in the CMP, the next issue is to determine how these components are connected and how they can communicate with each other. 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 is in charge of maintaining and organizing 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 execution 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 in case of data loss.

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. For more complex components such as processor cores, on the other hand, test results are broadcasted to the rest of the system to update the operating system about the state of the available computing elements. More frequent diagnostic tests lead to more reactive systems but also cause higher performance impact and diagnostic message proliferation. Thus their frequency is a design trade-off between extra traffic experienced in the system and reactivity to faults (analyzed in Section 5).

Diagnostic messages broadcasted to the system are collected by the local distributed managers, updating two software structures that maintain hardware state: a list of available hardware resources and a live interconnect graph. The first lists all hardware resources currently available in the CMP, and is used by the operating system to allocate hardware resources to the running applications. 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 is updated to allow the running applications to be remapped on the available resources and to trigger system reconfiguration.

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 [22, 27]. The notification-reaction paradigm that Cardio develops 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.

4. 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 [14]. 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. In Cardio, the discovery and reconfiguration proceeds depending on whether the failed unit is a core or a router link.

4.1 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) [5]. If the self-tests succeed, a checkpoint of the local state of the hardware is taken, as shown in the step 3 in Figure 2. Test outcomes are then enveloped in diagnostic messages that are 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 tests performed on the cores, more detailed information about faulty cores' capability can be provided. For instance, a core which floating point unit is not functional might still be reported as active, 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 require to be synchronized to allow for the system to be recoverable to a coherent state in case some other core failed. Differently from SafetyNet [27], 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 managers. 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. Note that 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. If some cores do not receive one or more diagnostic messages by the beginning of the next epoch, 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 new list of available cores from the diagnostic messages received. Moreover, the diagnostic messages require a synchronization mechanism to avoid 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 throughly 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, in our design we allow diagnostic messages to be broadcasted with a period which is half the checkpoint interval.

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 a messages to reach all cores in the system. Table updates for healthy hardware components can 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, it will not be reported as available and the other local resource managers will detect the failure at the beginning of their next self-test period.

4.2 Interconnect Monitoring

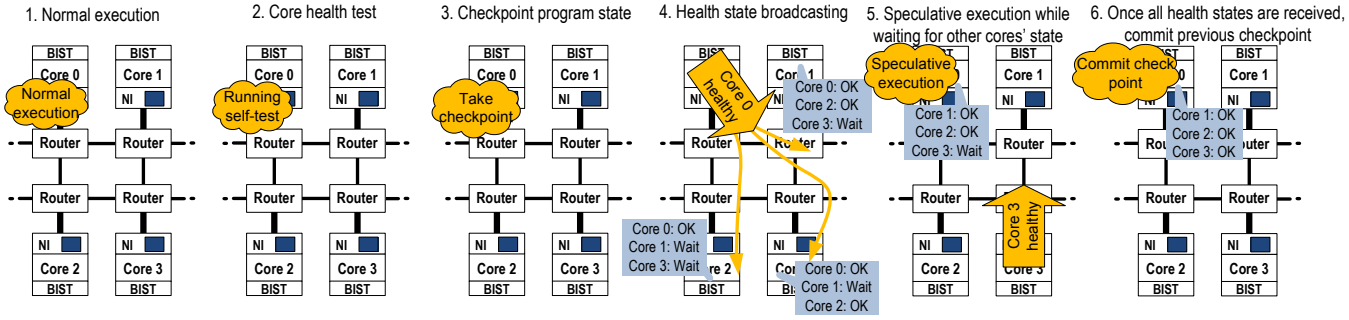

Figure 2: Core monitoring and recovery in Cardio. To maintain an updated state of the available cores in the system, Cardio relies on a five step sequence. 1) The cores perform their normal functions. 2) Core 0, independently from any other core, executes a self-test procedure to detect potential permanent 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.

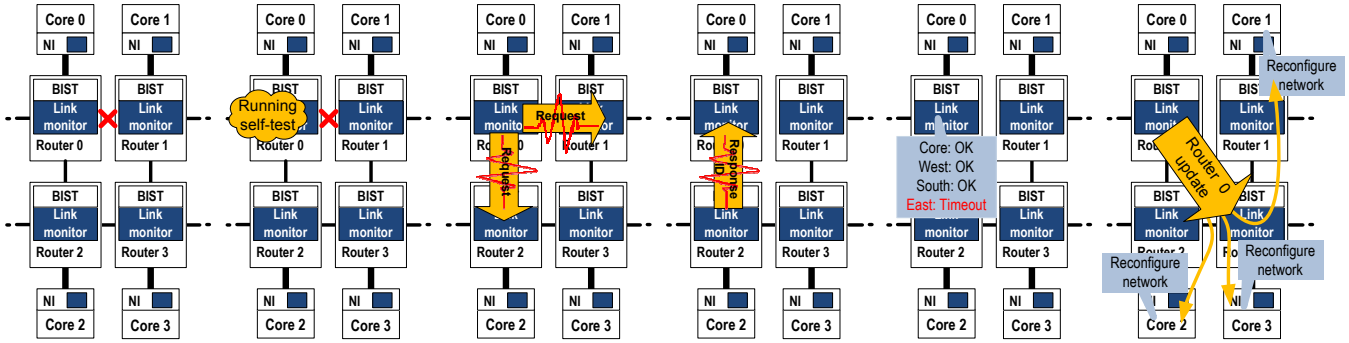

Figure 4: Dynamic interconnect management in Cardio. Self-discovery and reconfiguration in the interconnect are organized in five steps. To reduce the amount of extra traffic in the CMP, only topology *changes* are advertised. In the figure: 1) The NoC performs its normal functions. 2) Router 0 pauses 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 attaching, 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.

```
 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
```

Figure 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.

Correct functionality of the interconnect is vital for any digital design. Cardio can be successfully adopted in any NoC topology, and knowledge about the interconnect 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 [13]. For our on-chip dynamic discovery system, we decided to adopt a routing algorithm inspired by link-state solutions. These protocols converge more quickly and are more scalable than those based on distance vector, even though they are also more complex 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). The outcome of this first test determines if router's hardware can operate on each input and output link. The next step performed by the router is to dynamically discover which nodes are 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 heartbeats consist of a discovery request message to which recipients respond with a discovery response and a 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 due to

a fault), it drops all packets that were directed towards that link and broadcasts its updated local link table to the rest of the system (step 6 in Figure 4). Due to storage and performance constraints, the interval between interconnect checks should be at least a few thousand cycles long, because of its otherwise adverse impact on system performance, as discussed in our 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 manifest as available.

Since network failures might cause some packets in flight to be dropped, a retry mechanism is necessary to avoid communication loss. Cardio tackles runtime communication failures 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 all pending communications waiting for acknowledgment. These counter are incremented at every cycle and trigger time-out signals. In case of timeout, 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. The size of the acknowledgment buffers is a trade-off between storage and performance that we evaluate in our experimental evaluation in Section 5.

4.3 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 lightweight software layer is able to evaluate which resources are available in the device and how to access them.

The 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 can only reconstruct the topology of the region to which it belongs. Given an 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 some resource managers disagree on the available hardware components, a further negotiation among the components of the distributed resource manager can be 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, but the next checkpoint is not taken unless advertisements from all previously healthy cores have been received.

Note that cores and routers 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 the possibility that a fault causes a corrupted component to incorrectly advertise its state as correct. The probability of such event 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 takes, a special message is sent to all cores in the system to rollback to the previous synchronized checkpoint to prevent other processors from committing speculative executions 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.

4.4 Applicability

The distributed algorithms used by Cardio and the hybrid hardware/software system-level reliability solution proposed in this work are general concepts that can be extended to various CMP architectures. Cardio relies on the ability of hardware components' broadcast of diagnostic messages and on point-to-point communication to synchronize the distributed resource manager and reconfigure the healthy modules. Finally, to ensure Cardio's functionality, at least one general purpose core in the design must be able to execute the software routines needed to manage the hardware components at runtime.

5. EXPERIMENTAL RESULTS

We developed a distributed mechanism to manage and organize on-chip runtime faults. Our solution adds extra traffic due to the diagnostic messages exchanged by the self-checking hardware components. Therefore, 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 runtime impact on several application.

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 the number of extra communications exchanged in the system.

5.1 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 packet-switching system: interconnect components are cycle-accurate 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 can be dropped *(drop-packets)*, or the communication path can be blocked, stalling all packets at

the broken link's source *(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 [24].

Both uniform random traffic and traces from the SPECMPI benchmark suite [18] 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 are instrumented through the Tuning and Analysis Utilities to obtain traces of the communication patterns among the cores executing the applications [26]. 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 it would be when running the native application in its original form.

5.2 Acknowledgment Buffer Sizing

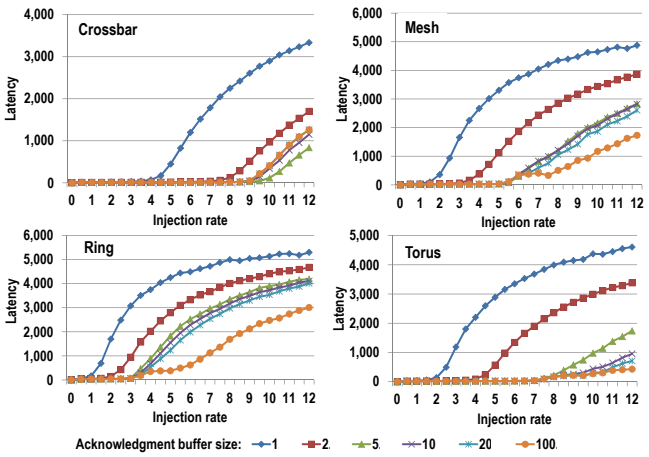

Figure 5: Packet latency vs. injection rate for different acknowledgment buffer sizes. Each curve represents a different acknowledgment buffer size as indicated in the legend. Packet buffers of size 10 provide the best trade-off between storage requirements and packet latency for all analyzed topologies.

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 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 4 shows the relation between the number of outstanding messages and average packet latency. Traffic injection rate is measured as the probability of each network interface injecting a new message 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 latency, while even doubling their size provides minimal benefits. Thus, in all subsequent experiments we set the acknowledgment buffer to store 10 outstanding packets.

5.3 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 amount of load: at a traffic injection rate of 5%. We selected this value 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 to bandwidth limitations. As shown in Figure 6, the trend is steeper for topologies such as mesh and ring, where 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.

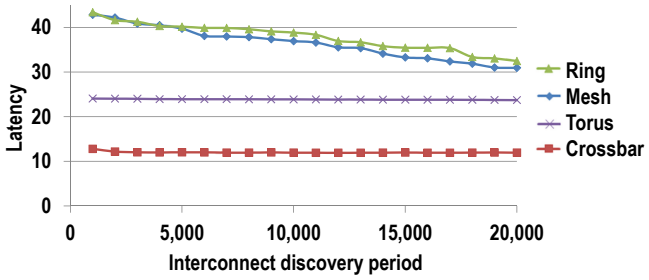

Figure 6: Packet latency vs. discovery period. The impact of the discovery period differs for different topologies: mesh and ring are more sensitive to variations due to a smaller network bisection.

5.4 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 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 stop at the link, clogging the network *(hold-packets)*. To provide insights on the dynamic behavior of Cardio, we analyze the system at 500-cycles intervals and report, on the Y-axis, *the average latency incurred by all packets* generated during each 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 [17]. 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 that of the interconnect components [4].

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, Cardio detects the problem and advertises the new 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 connected through a faulty link that may detect the problem at different times.

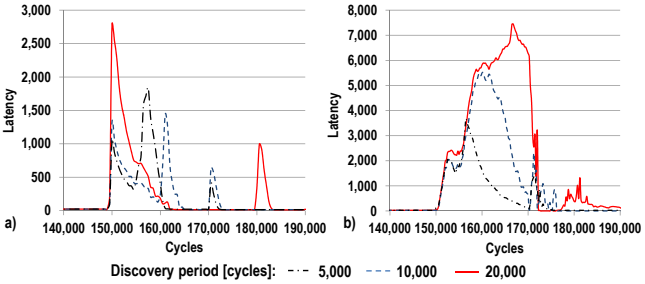

Figure 7: Effect of a runtime fault on a link. This graph plots the average time necessary for a packet to reach its destination; packet latency is averaged between all packets generated in a window of 500 cycles. In this scenarios the link is broken at cycle 150,000 and two fault models are considered: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

We then analyze the behavior under the *hold-packets* fault model. The impact under this model is more dramatic: a fault's effect is not limited to packets in transit between two nodes, but quickly 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 back 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.

5.5 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, even for very frequent discovery intervals is lower than 3% and almost uniform over all topologies. We report a very high performance impact of our solution on the *104.milc* benchmark for the mesh topology. This is because each core in that benchmark relies on very frequent and long data transfers with a single thread, mapped to the core on the top left corner of the mesh, which can access less bandwidth than the central nodes. The performance impact measured in this scenario is then particularly pessimistic.

Figure 8.b) shows the percentage of extra traffic introduced by our system. This graph also shows a rough estimate of the energy

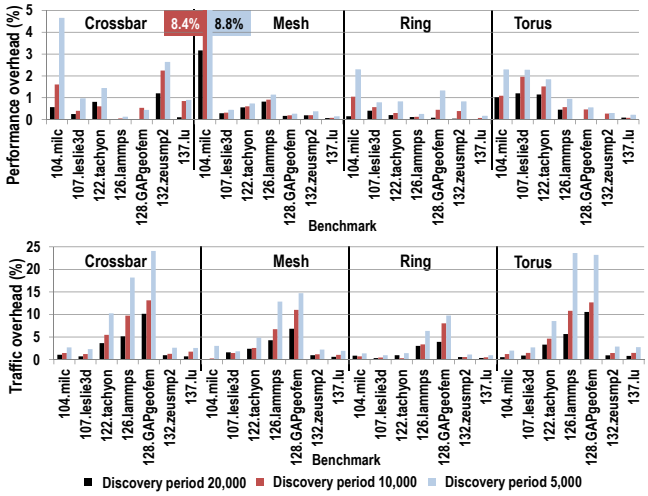

Figure 8: Performance impact and extra traffic (measured in message*hop) due to interconnect discovery on SPECMPI benchmarks. a) The performance impact for the considered applications is limited for most benchmarks (3%) and almost uniform over the 4 topologies. b) For most applications and topologies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

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 (such as *126.lammps* and *128.GAPgeofem*).

Other solutions for reliable interconnect, such as Immunet and Vicis, have no performance impact during fault-free operations, but they impose much larger silicon area overhead. Immunet requires three different routing table per node [23], while the overhead for Vicis is more than 40% of the design's baseline area [9]. Furthermore, Cardio provides global knowledge of hardware state to a middleware layer, thus enabling the development of tunable system level policies for hardware reliability.

5.6 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.

Moreover, the total storage requirements at each router is 6 bytes. Logic for handling the dynamic discovery protocol and reconfiguring the routing tables is also required. For comparison, each router in Immunet demands 28 bytes of additional storage. With larger interconnects both these trends grow linearly, so the benefits of Cardio are even more marked.

6. CONCLUSIONS

In this work we presented Cardio, a novel architecture hard-

ware/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. The additional storage required by Cardio in a 4x4 mesh CMP is only 345 bytes for each network interface and 6 bytes for each router.

7. 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, K. Brőndle, U. Brinkschulte, J. Henkel, W. Karl, T. Kőster, M. Wenz, and H. Wőrn. Digital on-demand computing organism for real-time systems. In *International Conference on Architecture of Computing Systems*, Mar 2006.

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

[4] 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 International Symposium on High-Performance Computer Architecture*, Feb. 2006.

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

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

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

[8] 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.

[9] 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.

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

[11] J. Howard and 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.

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

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

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

[15] Y. Li, M. Samy, and S. Mitra. CASP: Concurrent autonomous chip self-test using stored test patterns. In *Proc. of the Design, Automation and Test in Europe Conference*, Mar 2008.

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

[17] 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.

[18] 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. *International Journal of High Performance Computing and Networking*, Jan 2004.

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

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

[21] 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.

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

[23] 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.

[24] 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.

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

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

[27] 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 International Symposium on Computer Architecture*, May 2002.

[28] A. Strong, E. Wu, R.-P. Vollertsen, J. Sune, G. LaRosa, and T. Sullivan. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. Wiley Press, 2009.

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