Hindawi Publishing Corporation
VLSI Design
Volume 2016, Article ID 9767139, 17 pages
http://dx.doi.org/10.1155/2016/9767139

Research Article

Self-Healing Many-Core Architecture: Analysis and Evaluation

Arezoo Kamran and Zainalabedin Navabi

School of Electrical and Computer Engineering, University College of Engineering, University of Tehran, Tehran 1439956191, Iran

Correspondence should be addressed to Arezoo Kamran; akamran@ut.ac.ir

Received 27 December 2015; Accepted 30 May 2016

Academic Editor: Maurizio Palesi

More pronounced aging effects, more frequent early-life failures, and incomplete testing and verification processes due to time-to-market pressure in new fabrication technologies impose reliability challenges on forthcoming systems. A promising solution to these reliability challenges is self-test and self-reconfiguration with no or limited external control. In this work a scalable self-test mechanism for periodic online testing of many-core processor has been proposed. This test mechanism facilitates autonomous detection and omission of faulty cores and makes graceful degradation of the many-core architecture possible. Several test components are incorporated in the many-core architecture that distribute test stimuli, suspend normal operation of individual processing cores, apply test, and detect faulty cores. Test is performed concurrently with the system normal operation without any noticeable downtime at the application level. Experimental results show that the proposed test architecture is extensively scalable in terms of hardware overhead and performance overhead that makes it applicable to many-cores with more than a thousand processing cores.

1. Introduction

Impressive advances in emerging fabrication technologies have offered the capability of fabricating smaller, faster, and cheaper transistors and made integration of tens of billions of these devices on a single chip feasible. These capabilities have broadened the horizons to produce processors with more processing power by duplicating a processing core ten or even hundred times on a single chip. However deep submicron (DSM) manufacturing techniques face new challenges that can be categorized as (1) production yield drop, (2) early-life failure (infant mortality), (3) accelerated aging (wear-out), (4) soft errors (transient faults), (5) process variability, and (6) verification inefficiencies [1–3]. Because of all the challenges encountered in new fabrication technologies a major paradigm shift in all aspects of these technologies from design, verification, and fabrication to testing and software and application development is required.

Because of inadequate and nonaggressive burn-in testing, more pronounced aging effects, and incomplete testing and verification processes due to increased time-to-market pressure in new fabrication technologies, systems fabricated in these technologies may experience faults (including early defects or latent faults) and fail at any time in the field. So a one-time factory testing is not sufficient in future VLSI components. In fact success of new fabrication technologies heavily depends on integrating architectures and mechanisms into the design to deal with low yield and frequent in-field failure of components due to early-life failures or latent faults and guarantee long-life reliable operation of future chips.

Reliable operation of VLSI components can be achieved by redundancy-based techniques or test and reconfiguration. As the defect probability in new fabrication technologies increases, the overhead imposed by redundancy-based techniques becomes less tolerable. In fact the redundancy-based techniques in new defective fabrication technologies use a large amount of additional elements to protect on-chip components with redundant parts and not for improving the performance [4]. A possible solution to the reliability challenges of new fabrication technologies is self-test and self-reconfiguration with no or limited external control [5].

Considering various researches in this area, there is a need for a test mechanism that is applicable to many-core processors having a large number of processing cores, with a minimal hardware and performance overhead. This is what our research is devoted to. In this work we propose a scalable test architecture for online detection of permanent and intermittent faults in a homogeneous many-core processor. This architecture is a significant extension of our previous works [6, 7]. Using this test architecture, we will have a many-core processor that is autonomous and concurrent with the system normal operation, checks the health status of its processing cores, detects the faulty ones, and removes them from the system realizing a many-core processor architecture with self-healing capability.

The rest of this paper is organized as follows: In Section 2 related works and their pros and cons have been discussed. Section 3 presents details of the proposed test mechanism. In Section 4 various test strategies that can be realized in the proposed test architecture are discussed. Analysis on detection latency and probability of correct system recovery is presented in Section 5. Section 6 investigates experimental results, and finally Section 7 concludes the paper.

2. Related Works

More frequent in-field failures of chips due to silicon defects in recent extremely dense fabrication technologies have led to development of mechanisms and techniques to detect and isolate faulty components and reconfigure the chip to continue its correct operation. Constantinides et al. [8] have addressed this issue. They have added several new instructions to the processor instruction set architecture, called Access-Control Extensions (ACE), that provides access and control of the processor internal state via software. Special firmware periodically suspends the processor normal operation, stores the processor state, and applies the ACE instruction to the processor core. If the processor passes the test procedure, its sate is retrieved and the normal operation is continued. Otherwise, in presence of a permanent fault, this firmware can detect and locate the fault and trigger a reconfiguration mechanism. Because of the software nature of this mechanism it has a great flexibility to be modified in the field for more reliability or less performance overhead without need for any hardware changes. However, inclusion of ACE instructions may lead to performance degradation in highly optimized processing cores.

Li et al. [9] proposed a central system-level test hardware called CASP Test Controller for testing a shared memory chip multiprocessor with eight processing cores. The processor test hardware is connected to the processing cores via a crossbar switch. CASP Test Controller periodically isolates a processing core and applies stored test patterns to the core, concurrent with the normal operation of the rest of the system. Although this test mechanism is a good solution for chip multiprocessors with a few numbers of processing cores, the central nature of CASP Test Controller and the way that it is connected to the processing cores limit its scalability to CMPs with large number of processing cores.

Bernardi and Reorda [10] proposed a test architecture for manufacturing test as well as periodic testing of SoCs including cores equipped with BIST circuitry and IEEE 1500 wrappers. They have devised an Infrastructure IP named OTC (Online Test Controller) that is connected to the system bus and mapped in the address space as a common peripheral device, so it can be accessed and controlled by a CPU embedded in the SoC via software. CPU writes high level test commands into a RAM attached to OTC and determines the order and activation mode of the embedded cores. In response to the issued commands, OTC programs the BIST circuitry of each core to apply test and retrieve results via a dedicated test bus. This mechanism is properly applicable to cores that can well be tested by a BIST circuitry but is not suitable for testing of processing cores embedded in the SoC.

Rodrigues and Kundu [11] suggested a test mechanism that exploits the natural redundancy existing in a CMP to dynamically verify operation of the datapath portion of its processing cores. They have implemented an Inter-Core Queue (ICQ) as a communication channel between pairs of cores in the CMP. Whenever an instruction is retired on a core, it is pushed into the ICQ along with the operands and the result calculated by the core. The core residing on the other side of the ICQ pulls instructions from the ICQ whenever there is a vacant slot in its scheduling window. The instruction gets executed on the tester core and the result obtained by the tester core is compared with the previous result stored in the ICQ with the aid of some hardware circuitry. A mismatch in results indicates presence of fault. This mechanism only checks the datapath portion of the processing cores. In this mechanism, pairing of cores is static and whenever one core is detected as faulty, both of the coupled cores will be unusable.

Austin [12] used dynamic verification concepts and introduced a microarchitectural-based technique called DIVA (Dynamic Implementation Verification Architecture) enabling a complex processor to dynamically detect functional and electrical faults. DIVA is formed by splitting pipeline stages into two parts: DIVA core that fetches, decodes, and executes instructions and holds the results in a reorder buffer (ROB) and DIVA checker that augments the commit stage of a traditional out-of-order processor and verifies all the operations performed by the DIVA core. If a computation result is correct it is permitted to pass through the commit stage. Otherwise the DIVA checker fixes the result, flushes the pipeline, and restarts DIVA core at the instruction after the fixed instruction. Although DIVA checker that is an in-order pipelined processor with few interinstruction dependencies is simple compared with the DIVA core, its hardware overhead is not tolerable in simple processing cores of emerging many-core processors.

Benabdenbi et al. [13] proposed a software/hardware mechanism that is in charge of detection and reaction to hardware/software malfunction in a Multi-Processor System on Chip (MPSoC). They have assumed that cores are provided with status registers and counters measuring activity level with at least a serial access mechanism to read these registers. To realize the testing mechanism, they have proposed embedding of a dedicated hardware component that is in charge of periodically collecting data stored in the activity registers of the cores. The hardware component is driven by a software application that concurrently runs with the normal application and periodically compares the

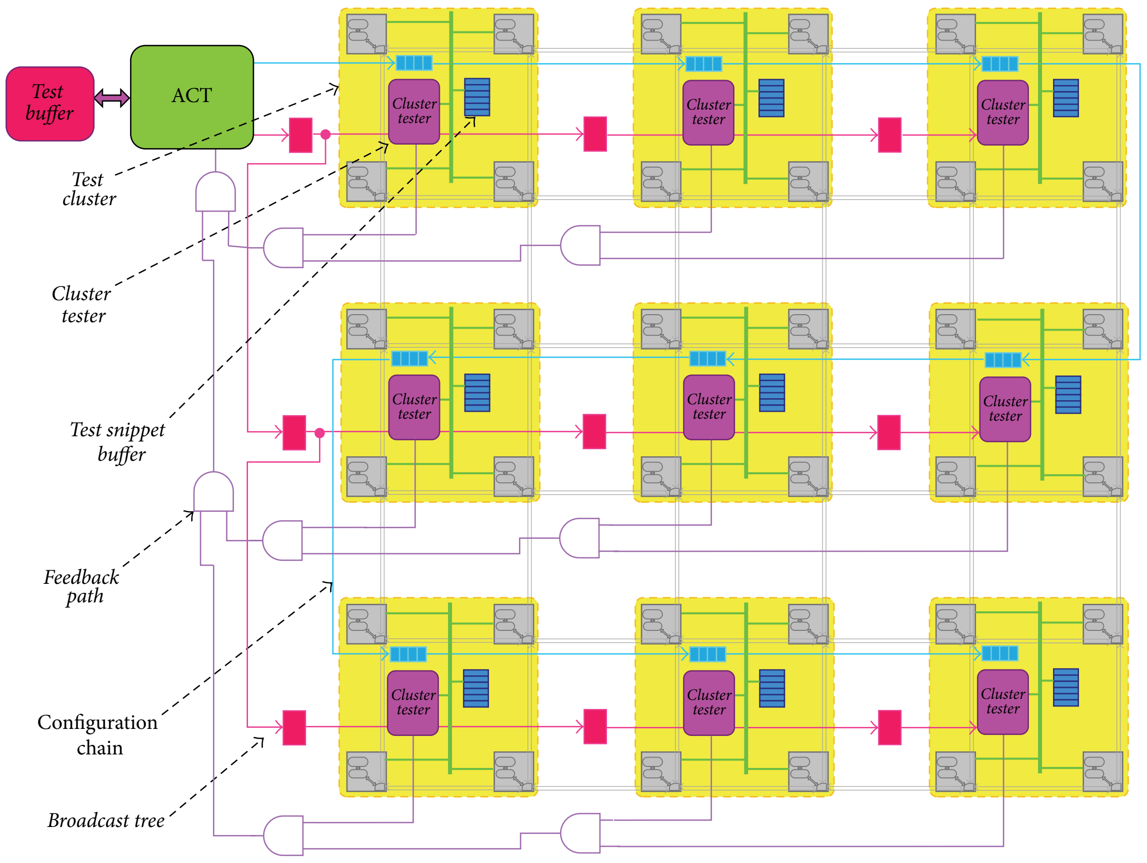


Figure 1: Test components incorporated into the many-core architecture.

activity counters with the expected values to detect if the functional application is still normally working. In this work, the detection mechanism is based on the activity counters, but the ability of the mechanism to detect hardware faults is not investigated. In fact, skepticism exists in detecting hardware faults through use of several high level activity counters.

Collet et al. [14] suggested an offline mechanism realizing a graceful degradable many-core processor. In this work a test program is stored in a local memory embedded in each node and the processing cores are tested by a software-based self-test (SBST) approach. Although SBST is a nonintrusive approach that enables at-speed testing of the processing cores with no overtesting, the assumption of local storage of the test program in each node imposes hardware overhead comparable with the hardware cost of simple processing cores.

3. Proposed Test Mechanism

In this section we introduce architectural details of a test distribution and execution mechanism for a homogeneous many-core processor architecture. We assume that the many-core processor is composed of several identical nodes each consisting of a processing core, several cache blocks, and some hardware facility for communications with other nodes. Although using routers arranged in a mesh topology is the most common communication structure in many-core processors, our proposed test architecture does not make any assumption about the communication infrastructure of the many-core processor. The focus of this work is on the testing of the processing cores, but it is easily extendable for testing of the other components of a node such as routers and cache controllers.

3.1. Proposed Test Mechanism: Hardware View.

3.1. Proposed Test Mechanism: Hardware View. The proposed test architecture consists of different components. Some of these parts such as *test buffer*, ACT (Autonomous Chip Tester), and *broadcast tree* are responsible for saving and broadcasting test stimuli among different nodes. Some other parts, namely, *cluster testers*, are locally situated near nodes and are responsible for receiving the test, isolating the corresponding cores, applying test to them, and identifying faulty ones. Figure 1 shows different parts of the test architecture that have been added to a many-core with mesh topology. In the following sections, different parts of the proposed test architecture are discussed in detail.

3.1.1. Test Buffer. *Test buffer* is an on-chip or off-chip nonvolatile memory that stores test patterns. Since all nodes are identical, only one copy of the test set is stored in the *test buffer* and is shared among all the nodes. The required

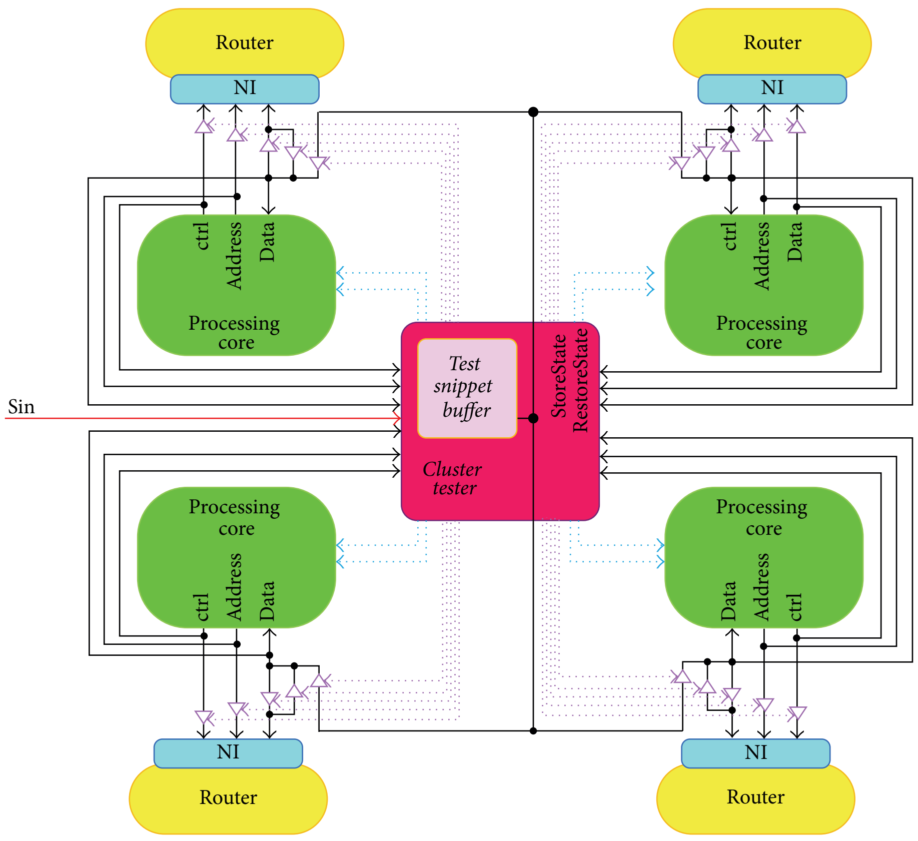


Figure 2: Overview of a test cluster.

nonvolatile memory to store test stimuli in the proposed architecture is less than 10 kbytes. This requirement is a small portion of the available storage in a modern chip.

3.1.2. Autonomous Chip Tester. Responsibility of *Autonomous Chip Tester* (ACT) is fetching test stimuli and test commands from the *test buffer*; broadcasting them among *test clusters* through the *broadcast tree*; and scheduling test procedures. In fact, ACT is a tiny processor with a limited ISA. It processes several test commands and implements the entire test procedure by executing those test commands. Different test commands that exist in the ISA of ACT are explained in Section 3.2.

3.1.3. Broadcast Tree. The *broadcast tree* of the proposed test architecture consists of an array of flip-flops arranged in a pipelined tree. Test commands and test data are injected into the tree by ACT and delivered to all processing cores via *broadcast tree*. The pipeline nature of *broadcast tree* can overcome the challenge of poor scalability of wire delays in new fabrication technologies. Compared with a serial pipeline, arrangement of flip-flops in a tree structure gives high scalability to *broadcast tree* in terms of latency of test distribution. As the number of nodes increases, the latency of delivery of test stimuli to the farthest nodes grows proportional to the square root of the number of nodes. In fact, the broadcast tree facilitates distribution of test data and test commands through the many-core architecture efficiently and independent of the functional communication infrastructure of the many-core architecture.

3.1.4. Test Cluster. A *test cluster* in a many-core processor (Figure 1) refers to several neighboring nodes, a *cluster tester*, and a small local buffer called *test snippet buffer*. Figure 2 shows an overview of a *test cluster* that consists of four processors. The responsibility of a *cluster tester* is to receive test commands and test data from ACT through the *broadcast tree* (see this in Figure 1), isolate the processing cores of the corresponding *test cluster*, apply test data to the processing cores, and identify and remove faulty processing cores in collaboration with ACT. The concept of *test cluster* provides better utilization of test hardware through sharing of test components among several adjacent nodes.

Test clusters are put together in *test groups*. Contrary to the nodes in a cluster, clusters in a group are not necessarily adjacent; that is, clusters in different areas of a many-core may be organized in one group. Looking top-down, a many-core has hierarchically several *test groups*, each group comprises several *test clusters*, and each cluster includes several adjacent nodes and shared test components. Each cluster has a specific *Group-Id* and *Cluster-Id*. In addition, each processing core in a cluster has its specific *Core-Id*. Consequently, each processing core can uniquely be identified by a (*Group-Id*, *Cluster-Id*, *Core-Id*) triplet. Upon receiving a test command that contains the three Ids, a *cluster tester* executes the command only if *Group-Id* and *Cluster-Id* fields of that command match those of its own or if the fields have a wildcard value. A field value is regarded as wildcard when all its bits are "1." A wildcard value in the *Group-Id*, *Cluster_Id*, or *Core_Id* fields of a test command implies that the command should be applied to all *test groups*, all *test clusters* in the activated *test groups*, and all the processing cores in the activated *test clusters*. For example, "*test-command 2, ALL, 5*" activates all *cluster testers* in group 2. Such *cluster testers* apply *test command* to the processing core with *Core_Id* = 5 in their corresponding cluster.

The main goal of clustering is hardware sharing and the main goals of grouping are controlling the degree of test concurrency, providing the possibility of testing many-cores with heterogeneous processing cores, and partitioning a many-core into different regions with different test frequencies. The concept of clustering and grouping is discussed in Section 4.2 in more detail.

3.1.5. Feedback Path. *Cluster testers* use a *feedback path* (see Figure 1) in order to announce the completion of test execution or to report latest health status of their corresponding clusters to ACT. There is a network of AND gates and flip-flops on this path, which combines feedback of all active *cluster testers* together, generates a cumulative result, and delivers that result to ACT.

3.1.6. Configuration Chain. Each *cluster tester* has a register for saving Id of the cluster and the group that it belongs to. *Configuration chain* (shown as a chain of registers in Figure 1) is a chain of *Cluster-Id* and *Group-Id* registers in the *cluster testers*. This chain allows ACT to change the partitioning scheme of the *cluster testers* in various groups.

3.2. Proposed Test Mechanism: Software View

3.2.1. Test Commands. In the proposed test environment, a test procedure is realized by executing a proper sequence of instructions supported by ACT (called test commands) that is referred to as a *test program*. These commands are used for test scheduling, test distribution, test application, checking the health status of the processing cores, and diagnosing and disabling processing cores that are found to be faulty. Major command types are summarized in Table 1.

3.2.2. Test Program. In the proposed test mechanism a test procedure is realized by ACT which fetches and executes a proper sequence of test commands, called a test program. The processor-nature of ACT and its special working style, which is reading and executing test commands, offers a considerable flexibility to the proposed test architecture. Considering specific conditions of the many-core in terms of acceptable level of performance reduction due to test, and the required reliability level, different test programs may be executed by ACT. That is, it is possible to make major changes in the test procedure only by modifying the test program, with no change in hardware parts of the test architecture.

```
NEW_TEST_CYCLE
   NEW_TEST_SNIPPET_ROUND
      TEST_SNIPPET_DATA        3C019c1a
      TEST_SNIPPET_DATA        34213667
      TEST_SNIPPET_DATA        3c02e2e5
      TEST_SNIPPET_DATA        3442c921
      TEST_SNIPPET_DATA        00220822
      TEST_SNIPPET_DATA        ac010000
      APPLY_TEST               ALL, ALL, ALL
   END_TEST_SNIPPET_ROUND
   NEW_TEST_SNIPPET_ROUND
      TEST_SNIPPET_DATA        3c014485
      TEST_SNIPPET_DATA        34210d84
      TEST_SNIPPET_DATA        3c024196
      TEST_SNIPPET_DATA        3442461b
      TEST_SNIPPET_DATA        00220820
      TEST_SNIPPET_DATA        ac010000
      APPLY_TEST               ALL, ALL, ALL
   END_TEST_SNIPPET_ROUND
END_TEST_CYCLE
CHECK_AND_DIAGNOSE          ALL, ALL, ALL
```

Pseudocode 1: Portion of a test program.

```
NEW_TEST_CYCLE
   NEW_TEST_SNIPPET_ROUND
      TEST_SNIPPET_DATA        3C019c1a
      TEST_SNIPPET_DATA        34213667
      TEST_SNIPPET_DATA        3c02e2e5
      TEST_SNIPPET_DATA        3442c921
      TEST_SNIPPET_DATA        00220822
      TEST_SNIPPET_DATA        ac010000
      APPLY_TEST               2, 5, 1
   END_TEST_SNIPPET_ROUND
   CHECK_AND_DIAGNOSE          2, 5, 1
   NEW_TEST_SNIPPET_ROUND
      TEST_SNIPPET_DATA        3c014485
      TEST_SNIPPET_DATA        34210d84
      TEST_SNIPPET_DATA        3c024196
      TEST_SNIPPET_DATA        3442461b
      TEST_SNIPPET_DATA        00220820
      TEST_SNIPPET_DATA        ac010000
      APPLY_TEST               2, 5, 1
   END_TEST_SNIPPET_ROUND
END_TEST_CYCLE
CHECK_AND_DIAGNOSE             2, 5, 1
```

Pseudocode 2: Portion of another test program.

Pseudocodes 1 and 2 show portions of two sample test programs. Both test programs broadcast two portions of test stimuli. Test program of Pseudocode 1 applies test stimuli to all processing cores in all *test clusters* of all *test groups*, and checking the test result is performed after applying both test portions. In contrast, test program of Pseudocode 2 applies test stimuli to processing cores with *Core_Id* = 1 situated in *test clusters* with *Cluster_Id* = 5, in *test groups* with *Group_Id* = 2,

Table 1: Major command types of ACT.

Command	Description
NEW_TEST_CYCLE	ACT announces a new *test cycle* *Cluster testers* perform initialization tasks
NEW_TEST_SNIPPET_ROUND	ACT announces a new *test round* *Cluster testers* perform initialization tasks
TEST_SNIPPET_DATA test data	ACT broadcasts *test data* *Cluster testers* store *test data* in *test snippet buffers*
APPLY_TEST (Group_Id, Cluster_Id, Core_Id)	ACT broadcasts *APPLY_TEST* command *Cluster testers* with matching (Group_Id, Cluster_Id) apply test to a processing core with matching Core_Id in their corresponding cluster
END_TEST_SNIPPET_ROUND	ACT announces the end of the current *test round*
END_TEST_CYCLE	ACT announces the end of the current *test cycle*
START_DIAGNOSIS	ACT announces the start of diagnosis phase
CHECK_AND_DIAGNOSE (Group_Id, Cluster_Id, Core_Id)	ACT asks specific *cluster testers* about results of testing. In the case of fault report, *ACT* broadcasts hierarchical *REPORT_STAUS* commands to find suspicious *test group*, *test cluster*, and processing core
APPLY_RETEST (Group_Id, Cluster_Id, Core_Id)	ACT requests retesting of a suspicious processing core The target *cluster tester* executes retesting command
CHECK_SUSPICIOUS_CUT (Group_Id, Cluster_Id, Core_Id)	ACT inquires about health status of a suspicious processing core after retesting The responsible *cluster tester* reports the status
DISABLE_CORE (Group_Id, Cluster_Id, Core_Id)	ACT announces disabling of a specific processing core The responsible *cluster tester* marks that core as disabled
REPORT_STATUS (Group_Id, Cluster_Id, Core_Id)	Targeted *cluster* tester(s) report the last status of its (their) corresponding cluster(s)

and after applying each portion of test stimuli the health status of tested processing cores is checked.

3.2.3. Test Snippets. In the proposed test mechanism, the entire test stimuli are not applied to a processing core at once. Instead, it is subdivided into smaller parts which are called *test snippets*. A *test snippet* is small enough that can be stored in a *test snippet buffer* which is a tiny buffer located in each *test cluster*.

3.2.4. Test Timing. Figure 3 shows a simple timing diagram describing timing details of a sample test procedure in the proposed test mechanism. As shown, the time line is divided into epochs. During longer starting portion of an epoch (the left-hand side), the many-core is performing its normal operation, but during the smaller portion of time at the end of an epoch, testing is performed. We call this portion of time a *test cycle*. Each *test cycle* consists of several *test rounds*. At the start of a *test round*, ACT broadcasts a *test snippet*. *Cluster testers* receive this *test snippet* and store it in their *test snippet buffers*. After that, if a *cluster tester* receives an *APPLY_TEST* command with matching Id or a wildcard value, it becomes active and applies its stored *test snippet* to all or a specific processing core in the *test cluster* based on the *Core_Id* field of the issued *APPLTY_TEST* command. In summary, during each *test round* one *test snippet* is broadcasted and applied to the processing cores. The whole test stimuli are applied in the form of consecutive *test snippets* during successive *test rounds* and are referred to as a *test cycle*.

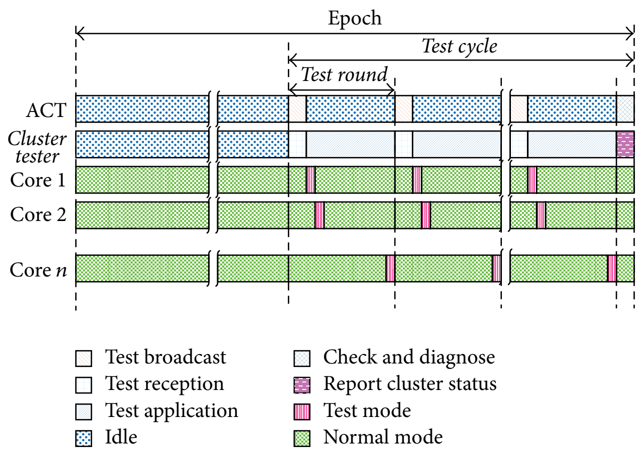


Figure 3: Timing details of a sample test procedure.

3.3. Cluster Tester and Test Procedure in Detail. Through broadcasting proper test commands, a *cluster tester* can be activated to perform various operations including

(i) receiving and storing test stimuli in the local *test buffer*,

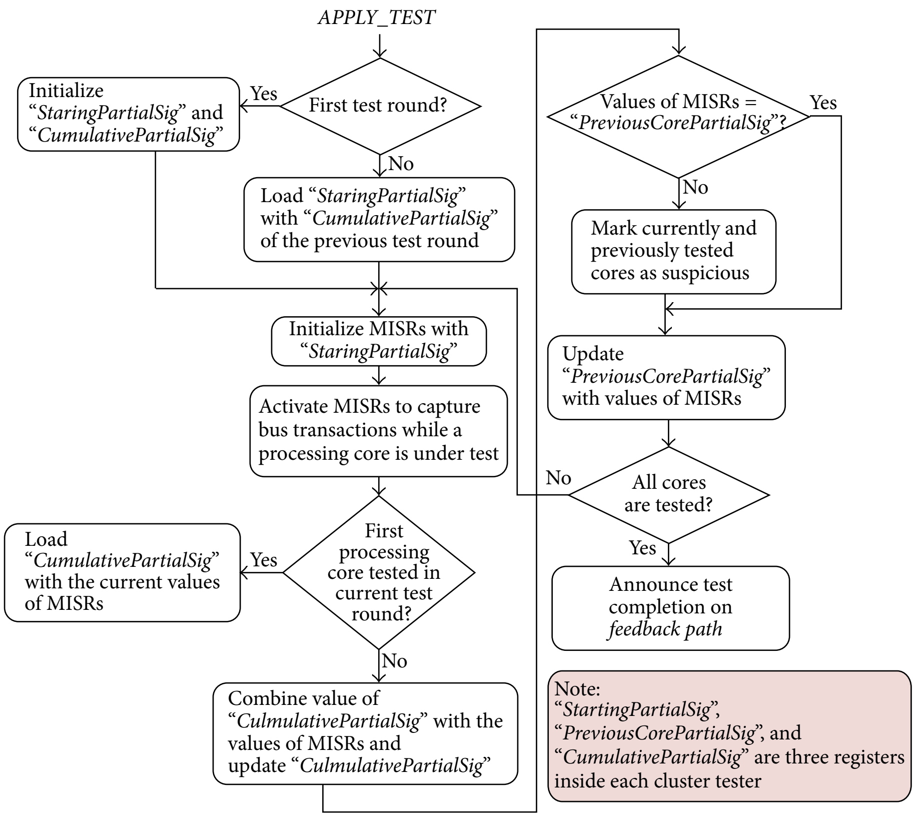

APPLY_TEST
First test round?
Yes
Initialize "StaringPartialSig" and "CumulativePartialSig"
No
Load "StaringPartialSig" with "CumulativePartialSig" of the previous test round
Initialize MISRs with "StaringPartialSig"
Activate MISRs to capture bus transactions while a processing core is under test
First processing core tested in current test round?
Yes
Load "CumulativePartialSig" with the current values of MISRs
No
Combine value of "CulmulativePartialSig" with the values of MISRs and update "CulmulativePartialSig"
Values of MISRs = "PreviousCorePartialSig"?
Yes
No
Mark currently and previously tested cores as suspicious
Update "PreviousCorePartialSig" with values of MISRs
All cores are tested?
No
Yes
Announce test completion on feedback path
Note: "StartingPartialSig", "PreviousCorePartialSig", and "CumulativePartialSig" are three registers inside each cluster tester

FIGURE 4: Flow of operations that a *cluster tester* performs when it is activated by an *APPLY_TEST* command.

(ii) application of test stimuli to one or all of the processing cores in the corresponding *test cluster,*

(iii) checking and reporting the health status of a specific processing core or all the processing cores in the cluster through *feedback path,*

(iv) marking a specific processing core as faulty.

A *cluster tester* issues proper signals to the processing cores of the corresponding *test cluster* to isolate them from the rest of the system and to push them to store or restore their architectural states before starting or after completion of testing, respectively. In addition, it consists of MISRs that generate signature of bus transactions when a processing core is under test. Figure 4 summarizes flow of operations that a *cluster tester* performs when it is activated by an *APPLY_TEST* command.

It should be mentioned when a processing core is isolated, all the other components of the corresponding *cluster* including cache blocks, scratchpad memories, switches, and routers remain in the normal mode and perform their normal operation. Consequently, only the state of the isolated processing core (and not the other components in a *cluster*) must be saved and retrieved before and after testing, respectively. Therefore not only do the overheads imposed by state storage and retrieval remain limited but also system correct operation during a test phase is guaranteed. Another point that should be considered is that test snippets are generated in a way that they are very short and can be executed in the order of 10 cycles. So each processing core is interrupted in a few cycles before it can switch back to normal mode, and the performance overhead due to irresponsiveness of a processing core that is being tested is negligible.

Our experimental results show that keeping separate signatures for each processing core imposes a large hardware overhead to the *test cluster*. So we have implemented a *cluster tester* in a way that it generates a cumulative signature of all bus transactions of all its tested processing cores. The idea of cumulative signature removes the burden of hardware overhead of distinct signatures but makes identification of faulty processing core(s) impossible. In the diagnosis phase when a faulty *test cluster* is retested to ensure that the reported fault has not been a transient one, all the processing cores in the cluster must be retested and this will affect diagnosis time. To cope with this problem, we have added a comparison logic to *cluster testers*. In each *test round* while testing of a processing core is completed, values of MISRs are compared with the partial signature generated by the previously tested processing core and any mismatch is captured as a sign of probable fault in the currently or previously tested processing cores. These processing cores are marked as suspicious and just suspicious cores will be retested in diagnosis phase. It should be mentioned that the comparison logic in fact realizes a Dynamic Dual Modular Redundant (DDMR) configuration between the processing cores of a *test cluster*. We call this *DMR* configuration a dynamic one because cores being compared are not fixed and are decided dynamically

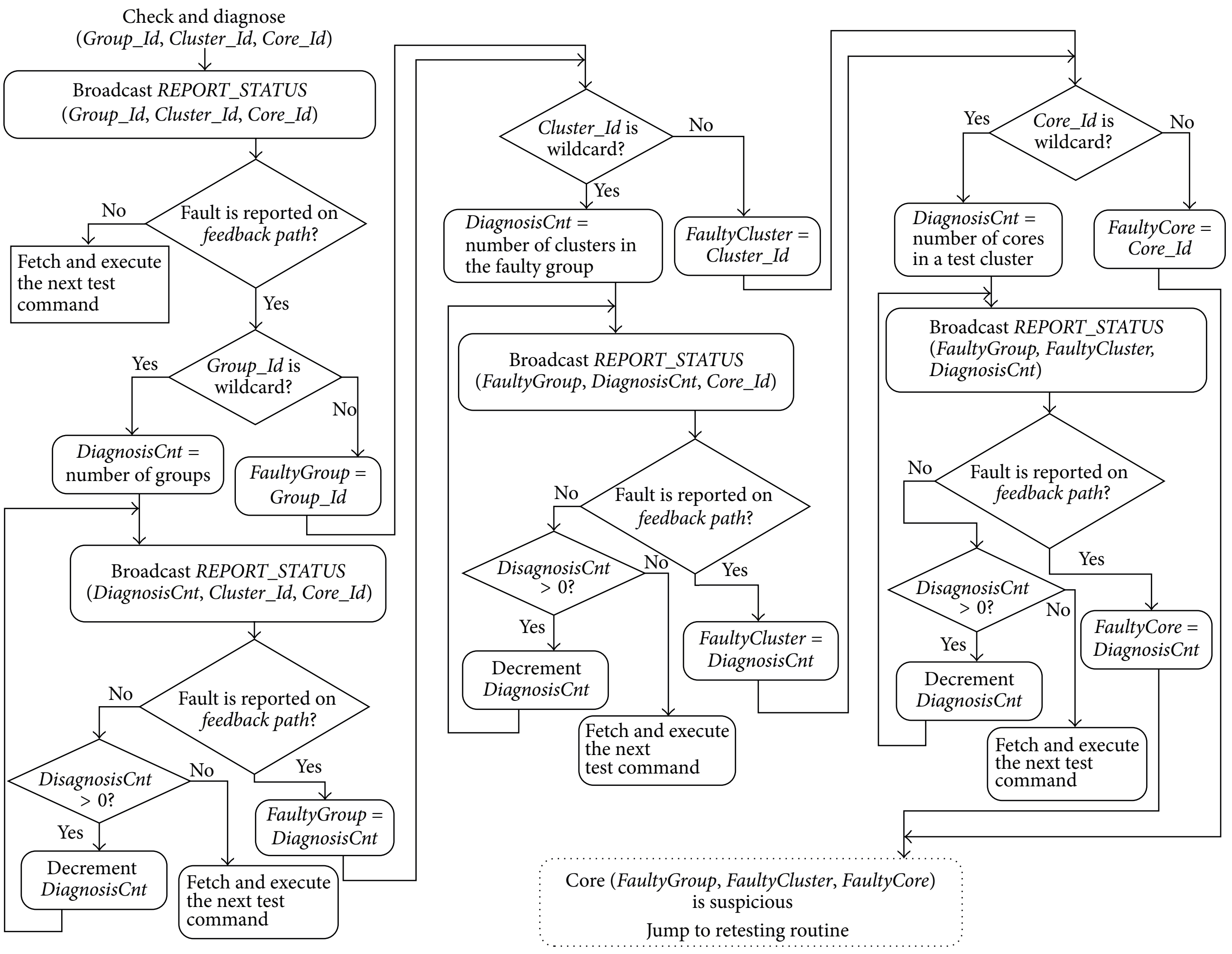

Check and diagnose (Group_Id, Cluster_Id, Core_Id)
Broadcast REPORT_STATUS (Group_Id, Cluster_Id, Core_Id)
Fault is reported on feedback path?
No
Fetch and execute the next test command
Yes
Group_Id is wildcard?
Yes
No
DiagnosisCnt = number of groups
FaultyGroup = Group_Id
Broadcast REPORT_STATUS (DiagnosisCnt, Cluster_Id, Core_Id)
Fault is reported on feedback path?
DisagnosisCnt > 0?
Decrement DiagnosisCnt
Fetch and execute the next test command
FaultyGroup = DiagnosisCnt
Cluster_Id is wildcard?
DiagnosisCnt = number of clusters in the faulty group
FaultyCluster = Cluster_Id
Broadcast REPORT_STATUS (FaultyGroup, DiagnosisCnt, Core_Id)
FaultyCluster = DiagnosisCnt
Core_Id is wildcard?
DiagnosisCnt = number of cores in a test cluster
FaultyCore = Core_Id
Broadcast REPORT_STATUS (FaultyGroup, FaultyCluster, DiagnosisCnt)
FaultyCore = DiagnosisCnt
Core (FaultyGroup, FaultyCluster, FaultyCore) is suspicious
Jump to retesting routine

Figure 5: Order of operations that ACT performs when it starts a diagnosis procedure.

at run time. In fact, in this mechanism dual processors are constructed dynamically based on the sequence at which processing cores are tested. So, when a processing core is marked as faulty and removed from the system, since it has no static pair, only that processing core is removed from the system and its couple is dynamically and automatically paired with another processing core in the *test cluster* without any external control. This is unlike static *DMR* scheme in which both cores should be made inactive.

3.4. ACT and Diagnosis Procedure in Detail. Figure 5 summarizes the order of operations that ACT performs when it starts a diagnosis procedure after receiving a fault report from the test structure's *feedback path*. In summary, ACT sends hierarchical *REPORT_STATUS* commands to find a probable faulty group (first column of Figure 5 flowchart), a faulty cluster in the faulty group (middle column), and a faulty processing core in the faulty cluster (rightmost column). In the first column only the *Group_Id* is being checked and the other fields are ignored. The middle column is reached when a group containing a faulty core is identified, and thus, the *Cluster_Id* is being checked. The last column is reached when a cluster containing a faulty core is identified, and the faulty core is being looked for. After finding the suspicious processing core, ACT jumps to the retesting routine (dotted box in Figure 5), issues *APPLY_RETEST* command to the suspicious *cluster,* and after retesting checks the result of retesting. If a fault is reported again, the suspicious processing core is assumed to have a permanent fault and ACT issues a proper *DISABLE_CORE* command. The target *cluster tester* marks this processing core as disabled after receiving this command. Otherwise if no fault is reported after retesting, the initial reported fault is assumed a transient one and ACT halts until the next *test cycle*.

4. Test Strategy

The proposed test architecture can adapt to various test strategies without changing the hardware. The processing nature of ACT as well as the flexible grouping mechanism gives this opportunity to a test engineer to decide on a suitable test procedure based on various considerations, which simply translate to various test programs with no need for any hardware change. Several considerations that must be taken

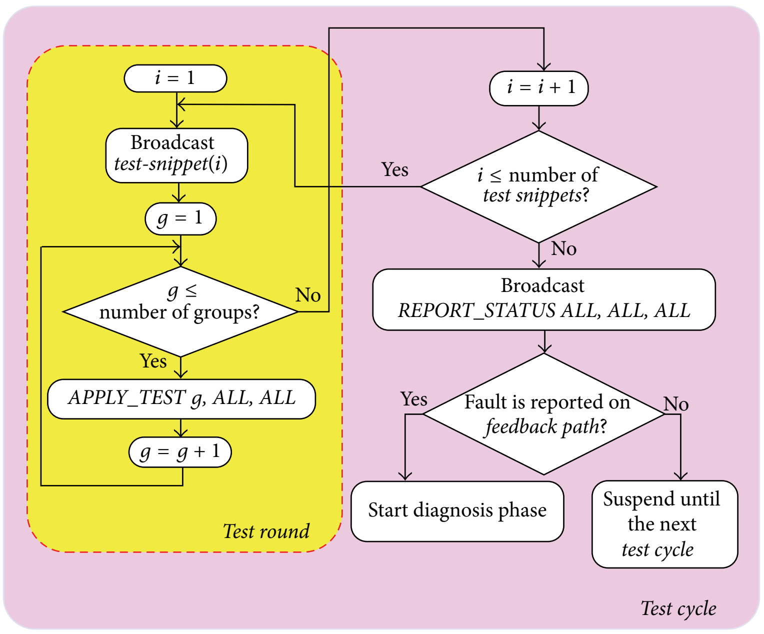

i = 1
Broadcast test-snippet(i)
g = 1
g ≤ number of groups?
No
Yes
APPLY_TEST g, ALL, ALL
g = g + 1
Test round
i = i + 1
i ≤ number of test snippets?
Yes
No
Broadcast REPORT_STATUS ALL, ALL, ALL
Fault is reported on feedback path?
Yes
No
Start diagnosis phase
Suspend until the next test cycle
Test cycle

Figure 6: *Ordinary testing* method.

into account while a test strategy is to be designed are listed below:

(i) There may be a need to classify processing cores into several groups with different testing requirements. Based on tasks that a group is running, the group may be more critical and needs to be tested more frequently or in more precision.

(ii) Many-core processors may contain heterogeneous processing cores. These processing cores may be categorized in different groups based on their type, and a proper test procedure can be designed for each group.

(iii) There are various degrees of performance degradation due to testing that a many-core can tolerate. Because of this, we can form different number of test groups, and at each instance of time only one test group is activated for testing (not performing its normal operation).

So based on different system conditions, suitable test procedures can be designed. In the next section we present a straightforward test strategy for a many-core with identical processing cores. This test strategy will be referred to as *ordinary testing* since it is our least sophisticated test strategy for identical processing cores with the same test requirements.

4.1. Ordinary Testing. We have the following assumptions in *ordinary testing*:

(i) The many-core processor consists of identical processing cores.

(ii) Processing cores should be tested with the same test frequency and test precision.

(iii) A limited percent of processing cores can be tested simultaneously. We call this parameter *TCP* (test concurrency percent).

We form several *test groups* each consisting of several *test clusters* in such a way that the number of *test clusters* belonging to each group is less than TCP × (total number of processing cores). The number of processing cores in each *test cluster*, number of *test groups*, and method of assigning *test clusters* to *test groups* are important issues that will be discussed in Section 4.2. At this point we assume that we have a good method for configuring the test architecture to achieve a minimum test time. Figure 6 presents details of our *ordinary testing* method.

The procedure begins by ACT broadcasting a *test snippet*. After that, it issues an *APPLY_TEST* command activating all *test clusters* of a specific *test group* and waits until the completion of test application. ACT repeats this procedure for all of the *test groups* one by one, until a *test round* is completed. This procedure is performed for all *test snippets* in successive *test rounds*. At the end of the final *test round*, ACT issues a *REPORT_STATUS* command targeting all *cluster testers* in all *test groups*. If all *cluster testers* report healthy status, ACT hibernates until the next *test cycle* is triggered; otherwise if a fault is reported, ACT starts the diagnosis procedure. Test time of *ordinary testing* can be calculated using

$$\text{Test-time}_{\text{ordinary}} = n_{\text{TS}} \times \Bigg(t_{\text{TS-Delivery}} + \sum_{i=1}^{G} \left(t_{\text{APPLY\_TEST}} + L(i) + t_{\text{TS-Exe}} \times S_{\text{Cluster}} + L(i) \right) \Bigg) + t_{\text{REPORT\_STATUS}} + 2 \times L_{\max}. \quad (1)$$

Parameters of (1) are described as follows:

n_{TS}: number of *test snippets*.

$t_{\text{TS-Delivery}}$: number of cycles to deliver a *test snippet*.

G: number of *test groups*.

$t_{\text{APPLY\_TEST}}$: number of cycles required to issue an *APPLY_TEST* command.

$L(i)$: number of cycles it takes a command to reach to the farthest *test cluster* in *test group i*.

$t_{\text{TS-Exe}}$: number of cycles required to apply a *test snippet* to a processing core.

S_{Cluster}: number of processing cores in a *test cluster*.

$t_{\text{REPORT\_STATUS}}$: number of cycles required to issue a *REPORT_STATUS* command.

$L_{\max}$: number of cycles it takes a command to reach to the furthest *test cluster*.

After some simplifications, (2) is obtained. In the next section we use this equation to find a good clustering and grouping scheme with the purpose of reducing test time:

$$\text{Test-time}_{\text{ordinary}} = n_{\text{TS}} \times \left(t_{\text{TS-Delivery}} + G \times \left(t_{\text{APPLY\_TEST}} + S_{\text{Cluster}} \times t_{\text{TS-Exe}} \right) + 2 \times \sum_{i=1}^{G} L(i) \right) + t_{\text{REPORT\_STATUS}} + 2 \times L_{\max}. \quad (2)$$

4.2. Clustering and Grouping. Test time of an *ordinary testing* mechanism can be achieved using (2). The values of many parameters in this equation directly depend on the physical dimensions of the many-core and cannot be changed. By ignoring these constant parameters, in order to minimize the test time of *ordinary testing* procedure, we have the following rules:

(1) The number of groups (G) should be minimized.

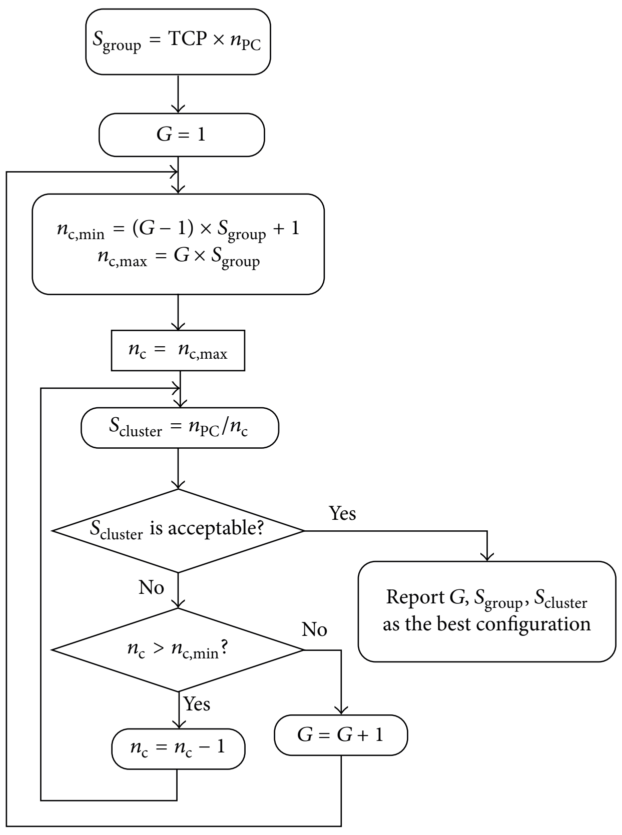
(2) The number of test clusters in each test group (S_{group}) must be equal to TCP $\times n_{\text{PC}}$ where n_{PC} is the total number of processing cores in the many-core. This allows test groups to execute the test procedure with the maximum acceptable concurrency.

(3) Clusters with similar $L(i)$ must be assigned to the same test group. This guarantees minimization of term $\sum_{i=1}^{G} L(i)$ in (2).

According to the above rules, algorithm of Figure 7 is proposed to find a test configuration with a minimum test time. Using this algorithm, G, S_{Cluster}, and S_{group} are determined.

As shown in flow diagram of Figure 7, we should find the minimum possible cluster size that keeps the number of test groups as low as possible in order to keep test cycle duration as short as possible. This is due to the sequential nature of test application in each test cluster and also in test groups. In other words, lower cluster sizes result in lower test cycles as long as the number of test groups does not grow. As shown in Figure 7, we start with the *number of test groups* (G) = 1 and find the minimum possible cluster size (S_{cluster}) that keeps the number of groups equal to G. As shown in Figure 13, with smaller cluster sizes, test hardware overhead increases. So we should determine the minimum cluster size that keeps the number of groups equal to G and at the same time, its test hardware overhead can be tolerated considering the maximum hardware budget. In the flow diagram of Figure 7, "S_{cluster} is acceptable?" means that can the test hardware overhead be tolerated assuming cluster sizes are equal to S_{cluster}? If so, the best configuration has been found; otherwise larger cluster sizes and then larger number of test groups must be considered.

Another problem is assigning *test clusters* to proper *test groups*. According to rule 3, *test clusters* with similar latencies should be assigned to the same *test group*. This is because latency of a *test group* is equal to the maximum latency



S_{group} = TCP × n_{PC}
G = 1
$n_{\text{c,min}} = (G-1) \times S_{\text{group}} + 1$
$n_{\text{c,max}} = G \times S_{\text{group}}$
$n_{\text{c}} = n_{\text{c,max}}$
$S_{\text{cluster}} = n_{\text{PC}}/n_{\text{c}}$
S_{cluster} is acceptable?
Yes
Report G, S_{group}, S_{cluster} as the best configuration
No
$n_{\text{c}} > n_{\text{c,min}}$?
No
Yes
$n_{\text{c}} = n_{\text{c}} - 1$
$G = G + 1$

n_{PC}: total number of processing cores
S_{group}: maximum number of clusters in each group
TCP: test concurrency percent
G: number of test groups
$n_{\text{c,min}}$: minimum number of test clusters assuming G groups
$n_{\text{c,max}}$: maximum number of test clusters assuming G groups
n_{c}: total number of test clusters
S_{cluster}: number of processing cores in each test cluster

Figure 7: Proposed algorithm to find a test configuration with a minimum test time.

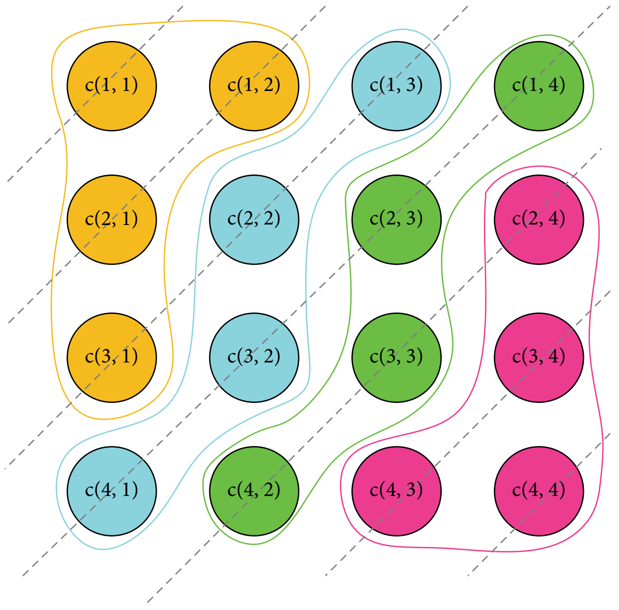


FIGURE 8: Assigning *test clusters* to *test groups*.

of its clusters. Figure 8 shows grouping of clusters of a many-core with 16 *test clusters* assuming that $S_{\text{group}} = 4$. Clusters located on the same diagonal line (see dotted lines in Figure 8) have the same latency. Assigning clusters to groups is started from $c(1,1)$ that is assigned to Group 1. Other clusters located on the nearest diagonal lines are added to Group 1 until S_{group} number of clusters is included in the group. This procedure continues for the next group until all *test clusters* are assigned. Several experiments have been performed to verify effectiveness of the proposed grouping and clustering scheme. Results of these experiments are presented in Section 6.

5. Fault Detection Latency and Probability of Correct System Recovery

In our proposed test mechanism, testing is performed at regular time intervals called epochs. Epoch duration can be adjusted to make a balance between performance overhead of the testing mechanism on the one hand and test efficiency, in terms of fault detection latency and probability of correct system recovery, on the other hand. In the next subsections, the system has been analyzed and *fault detection latency* and *probability of correct system recovery* are formulized. A test engineer can use these formulas to adjust epoch duration according to the intended values for *fault detection latency* and *probability of correct system recovery*.

5.1. Average Detection Latency of Permanent Faults

5.1. Average Detection Latency of Permanent Faults. Figure 9 shows a simple timing model for a many-core augmented with the proposed test mechanism. As shown in the figure, an epoch consists of two phases. In the first phase, the many-core is performing its normal operation. At the end of an epoch, testing mechanism is activated and processing cores

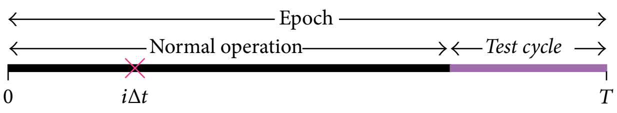


FIGURE 9: Timing model for a many-core *augmented* with the proposed test mechanism.

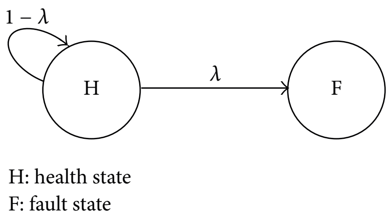


FIGURE 10: Markov chain of a system with permanent fault.

are tested. Let T be the epoch duration and T_{test} the test phase duration. Suppose that, at time t, many-core encounters a permanent fault. Latency of detecting this fault is $(T - t)$. The problem is finding the average value of detection latency (ADL).

Using Markov modeling and supposing that the average rate of occurrence of permanent faults in the system is λ, we can model the system with the Markov chain of Figure 10.

Using Markov analysis method, the probability of system being healthy or faulty is given by (3) and (4), respectively:

$$P_h(t) = e^{-\lambda t}, \tag{3}$$

$$P_f(t) = 1 - e^{-\lambda t}. \tag{4}$$

Based on this, the *average detection latency* of a permanent fault is given by (5). This equation shows how epoch duration can be decided based on desired *average detection latency* and existing rate of permanent fault occurrence (λ):

$$\begin{aligned}\text{ADL}_{\text{per}} &= \lim_{n\to\infty}\sum_{i=0}^{n}(T - i\Delta t)\,P_h(i\Delta t)\,(\lambda\Delta t)\\ &= \int_0^T (T-t)\,P_h(t)\,\lambda\,dt = \int_0^T \lambda(T-t)\,e^{-\lambda t}dt\\ &= T - \frac{1-e^{-\lambda T}}{\lambda}.\end{aligned} \tag{5}$$

5.2. Average Detection Latency of Intermittent Faults. A many-core similar to other digital systems may have intermittent faults. Intermittent faults may be active or inactive. Figure 11 shows Markov chain of a system with intermittent faults. Suppose that inactivation rate of an active intermittent fault is λ and activation rate of an inactive intermittent fault is μ.

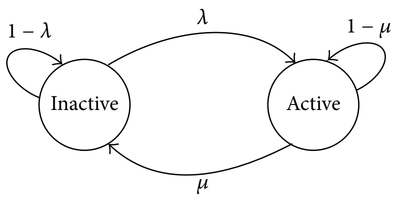


Figure 11: Markov chain of a system with intermittent fault.

Supposing that the system is at inactive state initially, the probability of system being at inactive state or active state at time t is given by (6) and (7), respectively:

$$P_{i,i}(t) = \frac{\lambda}{\lambda+\mu}e^{-(\lambda+\mu)t} + \frac{\mu}{\lambda+\mu}, \tag{6}$$

$$P_{i,a}(t) = \frac{\lambda}{\lambda+\mu}\left[1 - e^{-(\lambda+\mu)t}\right]. \tag{7}$$

In contrast, if the initial state of the system is supposed to be active state, then (8) are obtained:

$$\begin{aligned} P_{a,i}(t) &= \frac{\mu}{\lambda+\mu}\left[1 - e^{-(\lambda+\mu)t}\right], \\ P_{a,a}(t) &= \frac{\mu}{\lambda+\mu}e^{-(\lambda+\mu)t} + \frac{\lambda}{\lambda+\mu}. \end{aligned} \tag{8}$$

An intermittent fault is detected in the ith epoch if it is inactive at the end of all previous epochs and is active at the end of the ith epoch during the testing phase. Supposing that epoch duration is much greater than the test duration, the average detection latency of an intermittent fault is given by

$$\begin{aligned} \mathrm{ADL}_{\mathrm{int}} &= 0.5 \times \left[\sum_{i=1}^{\infty} P_{i,i}(T)^{i-1} \times P_{i,a}(T) \times iT\right] + 0.5 \\ &\quad\times \Bigg[P_{a,a}(T) \times T \\ &\quad+ \sum_{i=2}^{\infty} P_{a,i}(T) \cdot P_{i,i}(T)^{i-2} \times P_{i,a}(T) \times iT\Bigg]. \end{aligned} \tag{9}$$

According to Taylor series we have the following relation:

$$\sum_{i=1}^{\infty} ix^{(i-1)} = \frac{1}{(1-x)^2}. \tag{10}$$

Using (10), the following relations are obtained:

$$\begin{aligned} &\sum_{i=1}^{\infty} P_{i,i}(T)^{i-1} \times P_{i,a}(T) \times iT \\ &\quad= T \times P_{i,a}(T) \times \frac{1}{\left(1 - P_{i,i}(T)\right)^2} = \frac{T}{P_{i,a}(T)}, \\ &\sum_{i=2}^{\infty} P_{a,i}(T) \cdot P_{i,i}(T)^{i-2} \times P_{i,a}(T) \times iT \\ &\quad= T \times \frac{P_{a,i}(T) \cdot \left(1 + P_{i,a}(T)\right)}{P_{i,a}(T)}. \end{aligned} \tag{11}$$

Using (9) and (11) and after simplifications, the *average detection latency* of an intermittent fault is given by

$$\begin{aligned} &\mathrm{ADL}_{\mathrm{int}} \\ &= 0.5 \\ &\quad\times T\left[\frac{1}{P_{i,a}(T)} + P_{a,a}(t) + \frac{P_{a,i}(T) \cdot \left(1 + P_{i,a}(T)\right)}{P_{i,a}(T)}\right] \\ &= T \times \frac{1}{P_{i,a}(T)} = T \times \frac{\lambda+\mu}{\lambda\left(1 - e^{-(\lambda+\mu)T}\right)}. \end{aligned} \tag{12}$$

According to (5) and (12), one can decide on a proper epoch duration (T) to obtain a desired *average detection latency* for permanent and intermittent faults.

5.3. Probability of Correct System Recovery. If our proposed fault detection technique is accompanied by a proper checkpoint mechanism, the many-core can recover from permanent and intermittent faults. Recovery from permanent faults is deterministic whereas for intermittent faults it is an indefinite process.

As shown in Figure 9, a system may encounter permanent or intermittent faults at any time t. If a fault is detected during a testing phase and if the system is equipped with a proper checkpoint mechanism, the system is rolled back using the checkpoint captured at the start of the current epoch. If the occurring fault is a permanent fault, the system recovers completely. In contrast, if the system encounters an active intermittent fault during the testing phase of an epoch, rolling back the system to the previous checkpoint will not guarantee proper system recovery. Suppose that the intermittent fault is active during normal operation phases of epochs 1 to i but remains inactive during testing phases of all these epochs. If the intermittent fault becomes active in the testing phase of $(i + 1)$th epoch, it will be detected at the end of this epoch. But, if the system is rolled back using the checkpoint captured at the end of the ith epoch, system cannot recover from intermittent fault activities properly. The probability of proper system recovery from intermittent faults directly depends on epoch duration (T), activation rate of intermittent faults (λ), and inactivation rate of intermittent faults (μ). This can be calculated using (13). In the following equations we have used these notations:

$P_i(t)$: probability that intermittent faults remain inactive from time 0 to time t.

$P_a(t)$: probability that an intermittent fault is active at time t.

Consider probability of correct system recovery

$$= \sum_{i=0}^{\infty} \left[P_i(iT) \times \left[P_a((i+1)T) \mid (\text{intermittent faults are inactive from time 0 to time } iT)\right]\right] = \sum_{i=0}^{\infty} P_i(iT) \times P_a(T). \tag{13}$$

To calculate the first contributing term in (13), the system should be modeled using the Markov chain of Figure 10. So, according to (3) we have the following relation:

$$P_i(iT) = e^{-\lambda(iT)}. \tag{14}$$

To calculate the second contributing term in (13), the system should be modeled with the Markov chain of Figure 11. So using (6) and (7), the following relation is obtained:

$$P_a(T) = \frac{\lambda}{\lambda+\mu}\left[1 - e^{-(\lambda+\mu)T}\right]. \tag{15}$$

Using (13), (14), and (15), (16) is obtained:

$$\begin{aligned} &\text{probability of correct system recovery} \\ &= \sum_{i=0}^{\infty} e^{-\lambda(iT)} \times \frac{\lambda}{\lambda+\mu}\left[1 - e^{-(\lambda+\mu)T}\right] \\ &= \frac{\lambda}{\lambda+\mu}\left[1 - e^{-(\lambda+\mu)T}\right] \times \sum_{i=0}^{\infty}\left(e^{-\lambda T}\right)^i. \end{aligned} \tag{16}$$

Using (16) and Taylor series of (17), (18) is obtained:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}, \tag{17}$$

$$\begin{aligned} &\text{probability of correct system recovery} \\ &= \frac{\lambda}{\lambda+\mu}\left[1 - e^{-(\lambda+\mu)T}\right] \times \frac{1}{1-e^{-\lambda T}}. \end{aligned} \tag{18}$$

So, given specific values for λ and μ, and using (18), one can obtain a proper value for T (epoch duration), in a way that the many-core can recover from intermittent faults with a desired probability.

6. Experimental Results

We have developed a generic HDL model for a configurable homogeneous many-core architecture. This many-core can be configured in terms of the total number of processing cores, the number of processing cores in each *test cluster*, and the way in which different *test clusters* are grouped.

In our implementation platform we have chosen *Plasma MIPS* as the processing core [15]. This is consistent with the trend of using simpler processing cores in emerging many-core architectures. *Plasma* is a 32-bit RISC processor supporting a subset of *MIPS* instructions. It has three stages of pipeline with a forwarding logic, a complex memory controller, and a serial multiplier/divider module that makes its testing difficult. In the next sections, results of test generation, hardware overhead, test time, and performance overhead will be presented.

6.1. Test Snippet Generation. Although our proposed test mechanism can be configured for any sort of test stimuli, we have decided to test the processing cores by a software-based self-test (SBST) technique that tests a processor by a sequence of its native instructions. The SBST techniques are nonintrusive in nature and facilitate at-speed testing that is very important in the new fabrication technologies. Additionally, SBST techniques remove the overtesting problem associated with nonfunctional testing techniques. We have devised a *snippet*-oriented software-based self-test generation method to generate software-based *test snippets* for processing cores. Details of our proposed SBST technique are out of scope of this paper and are presented in another work [16].

Table 2 shows results of fault coverage of the generated test snippets on various components of *Plasma MIPS* processor. Table 3 compares some characteristics of software-based test generated for *Plasma MIPS* processor using our proposed method with two other works. As shown, the total fault coverage of our proposed method is slightly more than those reported by [17, 18]. In addition, we have generated a smaller test set with less execution time.

6.2. Hardware Overhead of the Proposed Test Architecture. We have synthesized different configurations of the proposed test architecture with different total number of processing cores and *test cluster* sizes. The hardware overhead of each instance has been calculated according to

$$\begin{aligned} &\text{Hardware overhead} \\ &= \frac{\text{Hardware cost of test components}}{\text{Hardware cost of all the processing cores}}. \end{aligned} \tag{19}$$

Figure 12 shows hardware overhead of ACT for different cluster sizes assuming that the many-core has 32 processing

Table 2: Fault coverage of our proposed test snippet generation method on *plasma MIPS*.

Module	# faults	FC (%)
ALU	2095	99.9
Multiplier/divider (sequential)	5818	96.8
Shifter	2248	99.9
Register file	18504	98.4
Control logic	824	92.3
Memory controller	2982	93.9
Bus multiplexer	1579	78.7
Program counter logic	1142	67.2
Pipeline	2242	98.3
Total	37434	96.1

Table 3: Comparison of SBST techniques.

Test characteristic	Proposed method	Method of [18]	Method of [17]
Total test volume (words)	2025	2329	2374
Total fault coverage (%)	96.1	95.6	95.3
Total execution time (cycles)	5150	11000	14427

cores. As shown in this figure, as the cluster size increases, ACT hardware overhead grows slightly. It should be mentioned that since ACT is shared among all the processing cores, hardware overhead of ACT decreases with the same rate that the number of processing cores increases.

Figure 13 shows total hardware overhead of all test components except ACT, for different configurations. It should be mentioned that for each cluster size we have investigated several configurations with different number of processing cores varying from a few processing cores to more than 1500. Our experiments show that the hardware overhead is almost independent of the total number of processing cores and cluster size is the dominating parameter determining hardware overhead of test architecture. Figure 13 reports average hardware overhead of different configurations for each cluster size.

6.3. Effectiveness of the Proposed Clustering and Grouping Scheme. We proposed a method for clustering and grouping in Section 4.2 that would be used for test time reduction. In order to show the effectiveness of our proposed method we have generated different many-core instances with random configurations regarding cluster size and grouping scheme. Figure 14 compares test time of these configurations with the configuration determined by the proposed grouping and clustering scheme. As shown in the diagrams, in all cases the proposed scheme leads to the best configuration in terms of test time.

6.4. Test Time. Figure 15 shows test time of many-cores with different number of processing cores. Clustering and grouping of all many-cores have been performed using the proposed method. For each many-core test time has been achieved for three different cases of TCP = 5%, TCP = 10%, and TCP = 50%, meaning that 5%, 10%, and 50% of the processing cores of the many-core can be tested simultaneously. Graphs of Figure 15 show that our proposed test mechanism is very scalable in terms of the number of processing cores. As the number of processing cores increases, test time grows with a little slope. Graphs show that when the number of processing cores increases more than 16 times (1600%), test time grows about 8.4%, 8.6%, and 12.8% for TCP = 5%, TCP = 10%, and TCP = 50%, respectively.

In [19] a software-based self-test technique has been proposed for shared memory chip multiprocessors (CMPs). In this work, processing cores are connected to memory banks using a crossbar switch. The largest CMP experimented in this work has eight processing cores. Table 4 compares test time of this technique with our proposed method. As shown, test time of our proposed test mechanism is more than that achieved in [19] when the number of processing cores is 4. But as the number of processing cores increases to 8, test time of our proposed test mechanism is better than that reported by

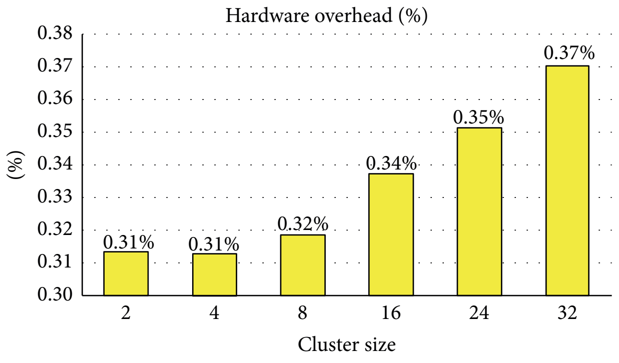


Figure 12: Hardware overhead of ACT for different cluster sizes.

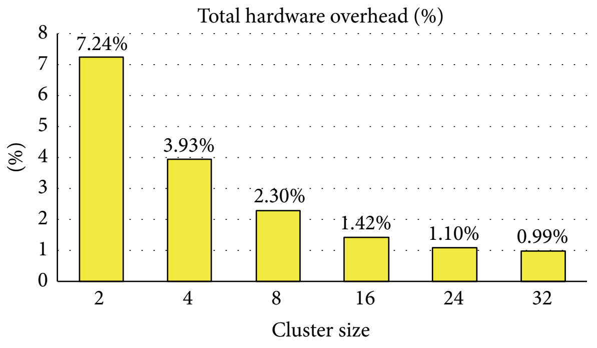


Figure 13: Total hardware overhead of all test components except ACT.

Table 4: Comparison of test time of our proposed method with [19].

Configuration		Test time (10^3 cycles)
4 processing cores	Quad 2 [19]	~93
	Proposed	157.232
8 processing cores	Octo 2 [19]	~166
	Proposed	158.324

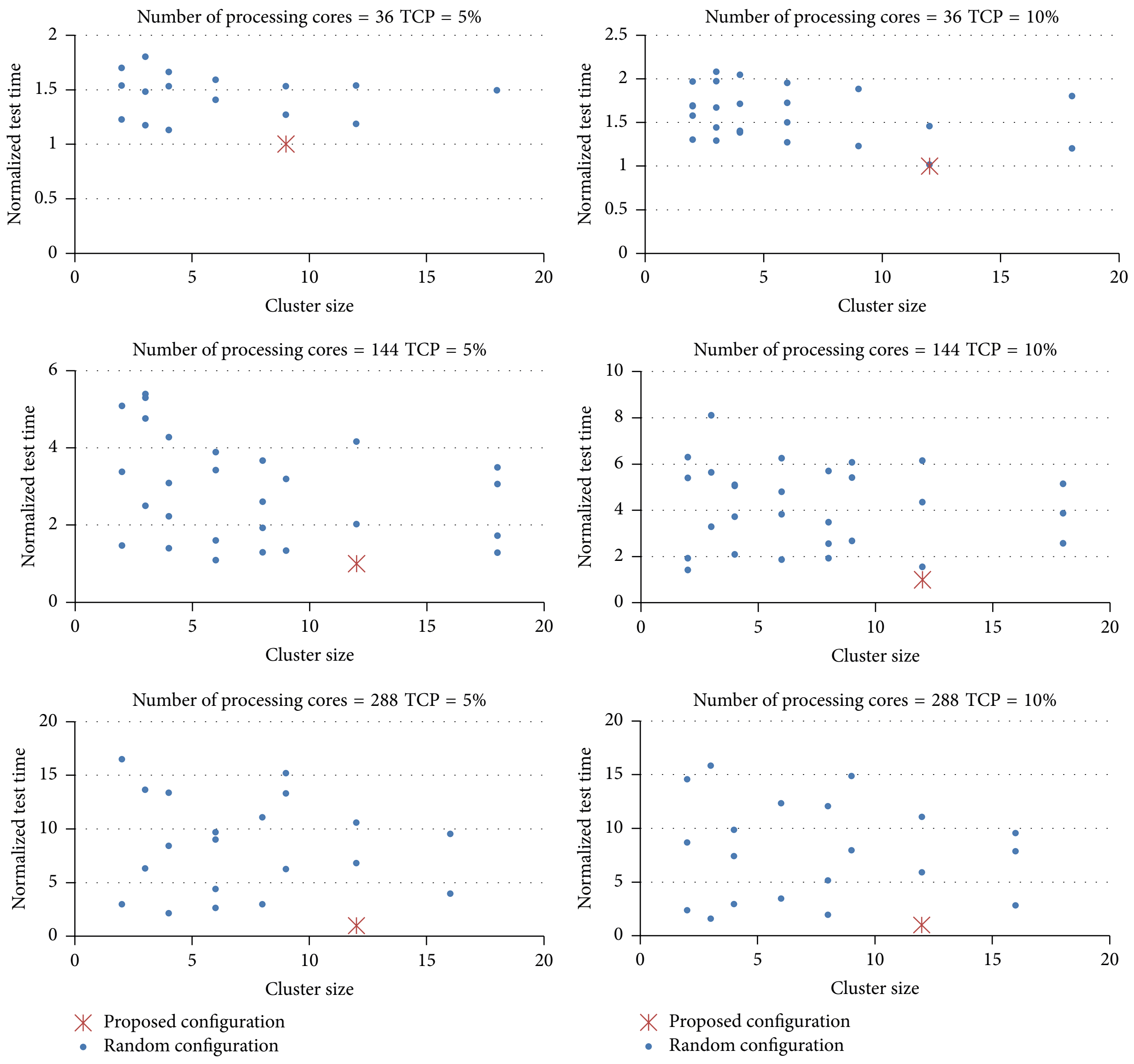


Figure 14: Comparison of test time of several random configurations with the configuration determined by the proposed grouping and clustering scheme.

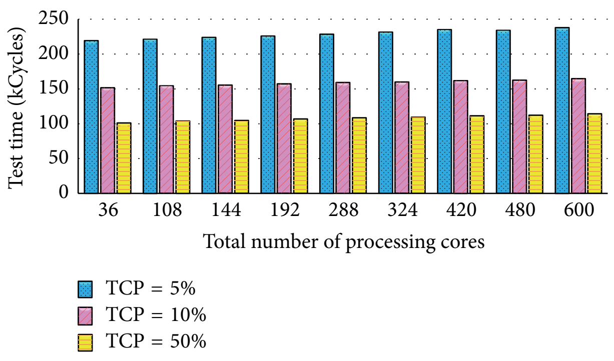


Figure 15: Test time of many-cores with different number of processing cores.

[19]. This is because of the scalability of our proposed method. Compared with [19], our proposed method can test larger number of processing cores with a slight increase in test time.

In a later work [20], a test program parallelization method has been proposed to accelerate online permanent fault detection in many-core architectures. Table 5 compares test time of this method with our proposed technique. As shown, our proposed test mechanism has a much shorter test time than [20]. Additionally, as the number of processing cores increases from 12 to 48, test time of our proposed test mechanism increases about 1.9% while it increases about 18.7% in [20]. This shows that our proposed method scales better for a larger number of processing cores. It should be mentioned that this test time reduction and better scalability

Table 5: Comparison of test time of our proposed method with [20].

Configuration		Test time (10^6 cycles)
12 processing cores	[20]	3.2
	Proposed	0.109844
48 processing cores	[20]	3.8
	Proposed	0.111896

Table 6: Comparison of performance overhead of our proposed method with [19].

Configuration		Peak performance overhead (%)	Average performance overhead (%)
4 processing cores	Quad 2 [19]	—	1.58
	Proposed	5.36	0.02
8 processing cores	Octo 2 [19]	—	7.8
	Proposed	5.16	0.02

Table 7: Comparison of performance overhead of our proposed method with [20].

Configuration		Peak performance overhead (%)	Average performance overhead (%)
12 processing cores	[20]	—	13.33
	Proposed	4.81	0.46
48 processing cores	[20]	—	15.83
	Proposed	4.72	0.47

are achieved at the cost of about 4% hardware overhead. Since our original test size is less than the test data of [19, 20], in order to have a fair compression, test times of our proposed mechanism presented in Tables 4 and 5 have been reported for test data with the same size as those of [19, 20].

6.5. Performance Overhead. According to Figure 9, testing is performed at the end of each epoch during a *test cycle*. Before the start of a test cycle, test components are inactive and the whole system is performing normal operation. So during this interval performance overhead is 0%. During a *test cycle*, test components become active and perform testing of the processing cores. So a slight decrease in performance occurs. It should be noticed that, during test broadcast phases of a *test cycle*, all the processing cores are performing their normal operation and performance of the system is not affected at all. Only when a processing core is isolated for testing, its normal operation is interrupted.

Tables 6 and 7 compare performance overhead of our proposed method with [19, 20], respectively. Epoch duration is assumed to be about 24 MCycles to be comparable with the results reported in [19]. In our proposed method, peak performance overhead is less than 5.4% and decreases when the total number of processing cores increases. Average performance overhead of our proposed method is much less than [19, 20] and increases slightly as the total number of processing cores increases. On the other hand, performance overhead of testing method proposed in [19, 20] increases considerably when the number of processing cores increases. Therefore, at the cost of about 4% hardware overhead, compared with [19, 20], our proposed method has a very low performance overhead and can scale considerably better for large number of processing cores.

7. Discussion and Conclusion

In this work we have proposed a scalable test architecture that is periodical and concurrent with the system normal operation, tests processing cores of a many-core, and detects faulty cores without any perceptible system-level downtime. This procedure is performed autonomously without any need for an external control facilitating self-healing of many-cores in cooperation with a proper checkpoint mechanism. Because of the processing nature of the central test controller and the flexible grouping and clustering scheme, the proposed test architecture is easily configurable to tune hardware overhead, availability, and reliability. Although in this work we have focused on test of processing cores, this test architecture can easily be extended to test the other parts of a many-core such as communication infrastructure as well as the memory subsystem. The main advantages of this work are as follows:

(1) Low hardware overhead: hardware overhead of all test components for cluster size of 4 is about 4%. By increasing the cluster size the hardware overhead reduces.

(2) Limited performance overhead: peak performance overhead during a *test cycle* is less than 5.4% whereas average performance overhead for epochs longer than 500 kCycles (that are very short epochs compared with a quantum time cycle) is less than 1%.

(3) Scalability with respect to test time and hardware overhead: experimental results show that when the number of processing cores is increased by more than 16 times (1600%), test time only grows about 8.4%, 8.6%, and 12.8% for TCP = 5%, TCP = 10%, and TCP = 50%, respectively. Additionally, hardware overhead is almost independent of the total number of processing cores, and cluster size is the dominant factor affecting hardware overhead. This provides the ability to test many-cores with more than a thousand processing cores.

(4) Tunability for desired reliability and availability: the proposed method can easily be tuned to make a balance between reliability (more test frequency or test precision), performance overhead, and availability.

(5) Extendibility to other components of a many-core architecture: the proposed test architecture can be extended for testing of other components such as routers. In addition BIST-mechanisms can be incorporated into *test clusters* and can be triggered under control of ACT to facilitate testing of memory segments and interconnections.

(6) Supporting independent test regions: the proposed test mechanism provides the capability of constructing various reliability domains or regions with different test frequencies or test precision using the concept of *test groups*.

Competing Interests

The authors declare that they have no competing interests.

References

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

[2] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The impact of technology scaling on lifetime reliability," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 177–186, IEEE, Florence, Italy, July 2004.

[3] S.-H. Huang, W.-P. Tu, C.-M. Chang, and S.-B. Pan, "Low-power anti-aging zero skew clock gating," *ACM Transactions on Design Automation of Electronic Systems*, vol. 18, no. 2, article 27, 2013.

[4] J. H. Collet, M. Psarakis, P. Zajac, D. Gizopoulos, and A. Napieralski, "Comparison of fault-tolerance techniques for massively defective fine- and coarse-grained nanochips," in *Proceedings of the 16th International Conference on Mixed Design of Integrated Circuits & Systems (MIXDES '09)*, pp. 23–30, Lodz, Poland, June 2009.

[5] P. Zajac and J. H. Collet, "Production yield and self-configuration in the future massively defective nanochips," in *Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT '07)*, pp. 197–205, September 2007.

[6] A. Kamran and Z. Navabi, "Homogeneous many-core processor system test distribution and execution mechanism," in *Proceedings of the 19th IEEE European Test Symposium (ETS '14)*, pp. 1–2, Paderborn, Germany, May 2014.

[7] A. Kamran and Z. Navabi, "Online periodic test mechanism for homogeneous many-core processors," in *Proceedings of the IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC '13)*, pp. 256–259, IEEE, Istanbul, Turkey, October 2013.

[8] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "A flexible software-based framework for online detection of hardware defects," *IEEE Transactions on Computers*, vol. 58, no. 8, pp. 1063–1079, 2009.

[9] Y. Li, S. Makar, and S. Mitra, "CASP: concurrent autonomous chip self-test using stored test patterns," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 885–890, March 2008.

[10] P. Bernardi and M. S. Reorda, "A new architecture to cross-fertilize on-line and manufacturing testing," in *Proceedings of the 20th Asian Test Symposium (ATS '11)*, pp. 142–147, IEEE, New Delhi, India, November 2011.

[11] R. Rodrigues and S. Kundu, "An online mechanism to verify datapath execution using existing resources in chip multiprocessors," in *Proceedings of the 20th Asian Test Symposium (ATS '11)*, pp. 161–166, IEEE, New Delhi, India, November 2011.

[12] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '32)*, pp. 196–207, November 1999.

[13] M. Benabdenbi, F. Pecheux, and E. Faure, "On-line test and monitoring of multi-processor SoCs: a software-based approach," in *Proceedings of the 10th Latin American Test Workshop (LATW '09)*, pp. 1–6, Rio de Janeiro, Brazil, March 2009.

[14] J. H. Collet, P. Zajac, M. Psarakis, and D. Gizopoulos, "Chip self-organization and fault tolerance in massively defective multicore arrays," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 2, pp. 207–217, 2011.

[15] Plasma CPU Model, http://opencores.org/project,plasma.

[16] A. Kamran and Z. Navabi, "Hardware acceleration of online error detection in many-core processors," *Canadian Journal of Electrical and Computer Engineering*, vol. 38, no. 2, pp. 143–153, 2015.

[17] N. Kranitis, A. Paschallis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461–475, 2005.

[18] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 88–98, 2005.

[19] A. Apostolakis, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Software-based self-testing of symmetric shared-memory multiprocessors," *IEEE Transactions on Computers*, vol. 58, no. 12, pp. 1682–1694, 2009.

[20] M. Kaliorakis, M. Psarakis, N. Foutris, and D. Gizopoulos, "Accelerated online error detection in many-core microprocessor architectures," in *Proceedings of the IEEE 32nd VLSI Test Symposium (VTS '14)*, pp. 1–6, April 2014.