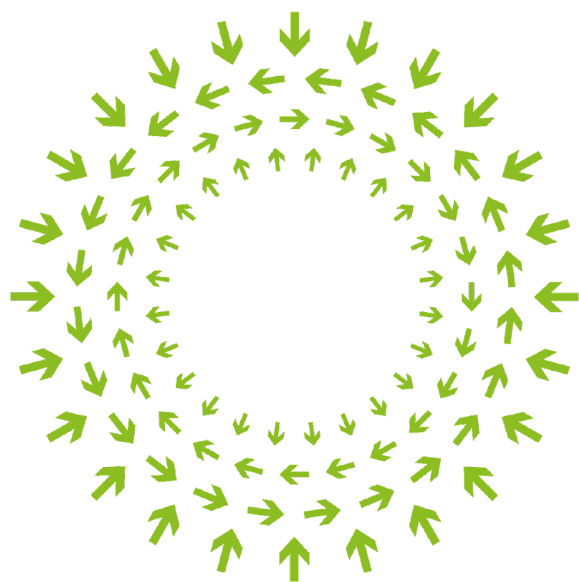




OPEN

Compute Project

Server Component Resilience

Revision 1.0
September 27, 2024

Contributors:

Bharath Parthasarathy and Amber Huffman, Google
Harish Dattatraya Dixit and Tejasvi Chakravarthy, Meta Platforms Inc
Rob Chappell, Microsoft
Vilas Sridharan and Sankar Gurumurthy, AMD
Thiago Macieira, Intel
Lisa Minwell, ARM
Lidia Warnes, NVIDIA

Table of Contents

Glossary of Terms	4
1. License (Resilience Workstream)	4
1.1. Open Web Foundation (OWF) CLA	4
1.2. Acknowledgements	5
2. Compliance with OCP Tenets	5
2.1. Openness	5
2.2. Efficiency	5
2.3. Impact	6
2.4. Scale	6
2.5. Sustainability	6
3. Version Table	7
4. Scope	7
4.1. In Scope	7
4.2. Out of Scope	8
5. Definitions	8
5.1 SDC-causing Computing system definition	8
6. Introduction to Information Exchange Specifications	8
Information Exchange Format:	9
7. Test Input Specification	9
8. Test Output Specification	10
9. Part History Specification	12
10. Example of Test Output and Part History	13
11. Metrics	15
11.1 Test Environment:	15
11.2 Test Efficiency	16
12. Test Framework and Test Flow	19
12.1 Pool access	19
12.2 Anatomy of a test	19
12.3 Test frameworks	20
12.3.1 Data Center GPU Manager	20
12.3.2 OpenDCDiag	21
12.3.3 Open Field Health Check	22
13. References	23
Appendix A - Checklist for IC approval of this Specification	24

Glossary of Terms

This section provides the glossary used in this specification. Note that it is not organized in alphabetical order but in sequential order to best understand the terms and definitions as they flow through the document.

SDC/ SDE = Silent Data Corruption/ Silent Data Error

SLT = System-Level Test

VM = virtual machine

Q-pool = quarantine pool

Fleet = Data Center

1. License (Resilience Workstream)

1.1. Open Web Foundation (OWF) CLA

Contributions to this Specification are made under the terms and conditions set forth in Open Web Foundation Modified Contributor License Agreement (“OWF CLA 1.0”) (“Contribution License”) by:

AMD, ARM, Google, Intel, Meta, Microsoft, NVIDIA

Usage of this Specification is governed by the terms and conditions set forth in **Open Web Foundation Modified Final Specification Agreement (“OWFa 1.0.2”) (“Specification License”)**.

You can review the applicable OWFa1.0 Specification License(s) referenced above by the contributors to this Specification on the OCP website at <http://www.opencompute.org/participate/legal-documents/>. For actual executed copies of either agreement, please contact OCP directly.

Notes:

- 1) The above license does not apply to the Appendix or Appendices. The information in the Appendix or Appendices is for reference only and non-normative in nature.

NOTWITHSTANDING THE FOREGOING LICENSES, THIS SPECIFICATION IS PROVIDED BY OCP "AS IS" AND OCP EXPRESSLY DISCLAIMS ANY WARRANTIES (EXPRESS, IMPLIED, OR OTHERWISE), INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, OR TITLE, RELATED TO THE SPECIFICATION. NOTICE IS HEREBY GIVEN, THAT OTHER RIGHTS NOT GRANTED

AS SET FORTH ABOVE, INCLUDING WITHOUT LIMITATION, RIGHTS OF THIRD PARTIES WHO DID NOT EXECUTE THE ABOVE LICENSES, MAY BE IMPLICATED BY THE IMPLEMENTATION OF OR COMPLIANCE WITH THIS SPECIFICATION. OCP IS NOT RESPONSIBLE FOR IDENTIFYING RIGHTS FOR WHICH A LICENSE MAY BE REQUIRED IN ORDER TO IMPLEMENT THIS SPECIFICATION. THE ENTIRE RISK AS TO IMPLEMENTING OR OTHERWISE USING THE SPECIFICATION IS ASSUMED BY YOU. IN NO EVENT WILL OCP BE LIABLE TO YOU FOR ANY MONETARY DAMAGES WITH RESPECT TO ANY CLAIMS RELATED TO, OR ARISING OUT OF YOUR USE OF THIS SPECIFICATION, INCLUDING BUT NOT LIMITED TO ANY LIABILITY FOR LOST PROFITS OR ANY CONSEQUENTIAL, INCIDENTAL, INDIRECT, SPECIAL OR PUNITIVE DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS SPECIFICATION, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND EVEN IF OCP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.2. Acknowledgements

The Contributors of this Specification would like to acknowledge the following companies for their feedback:

TBD

2. Compliance with OCP Tenets

2.1. Openness

The SDC specification aims to remove barriers by providing standardized information exchange formats and an open framework in addressing SDC given the diversity of chips in the data center. There are three principles that enable openness, collaboration and contribution from the entire community.

1. Open Frameworks to advance the state of the art in SDC testing, enabling the community to make contributions on detection methods.
2. Standard format for both input and output to enable open sharing and collaboration.
3. Create communication pathways and transparency to accelerate our collective work

2.2. Efficiency

Contributes to the reduction of test infrastructure costs by providing a standardized information exchange format and open framework for faster and better SDC detection techniques. Failing to

do it will resort to massive redundancy, which will hurt energy efficiency and performance of data centers. This Promotes energy efficient testing methodology by implementing optimized test content and efficient coverage of underlying hardware.

2.3. Impact

We are developing standardized formats and behaviors for sharing information about defective parts. This will enable researchers to develop better ways to protect data centers and open up new areas of research in hardware, software, and theory. In the past, experts in these fields have not been able to collaborate effectively due to limited data sharing. However, this is no longer sustainable. Standardizing data sharing will allow us to collaborate and create better solutions for data centers. This will enable us to codesign solutions across the system stack through a global supply chain, which will achieve the transformative impact that the Open Compute Project seeks.

2.4. Scale

SDC is an industry-wide issue impacting all platforms deployed threatening business workload and user data. The reliability of underlying hardware is critical to the future of artificial intelligence and machine learning. Furthermore, a single entity cannot solve the entirety of the problem considering how complex of a challenge it is and hence bringing the whole industry and academic community is key. Our contribution meets this scale tenet by defining standard information exchange formats and frameworks that enable adoption of new solutions and better detection methods through a global supply channel, accelerates reliable time-to-market advantage of technology, and supports ongoing innovation as well as legacy infrastructure.

2.5. Sustainability

Nothing can be sustainable if the fundamental tenet of computing is in jeopardy. So, any effort towards sustainability must overcome the SDC challenges. By ensuring there is a standardized format of information exchange and open framework to advance the state of art in SDC detection and mitigation across a wide variety of hardware, SDC specification can help improve the reliability and sustainability for future data center infrastructure.

3. Version Table

Date	Version #	Author	Description
June 30, 2023	0.3	See front page	Defines Scope and Inputs & Outputs for a test.
Sep 9, 2024	1.0	See front page	Added metrics and test framework sections

4. Scope

4.1. In Scope

As part of the [Open Compute Project](#) (OCP), in October, 2022 we established a Resilience working group (under the Hardware Management Project) focused on Silent Data Corruption (SDC) to create a community to address hardware-induced Silent Data Corruption challenges [Ref]. The scope of this initiative includes CPUs, GPUs, and other hardware accelerators. The founding members of this working group are AMD, ARM, Google, Intel, Meta, Microsoft, and NVIDIA.

The high level goals of the working group include:

1. Drive solutions and best practices that prevent and detect SDCs.
2. Create awareness about SDC challenges across the computing community.
3. Partner & engage with the academic community to actively address growing SDC challenges.

This specification document is intended to standardize test inputs, outputs, metrics, and formats to allow the industry and academia to collaborate on development of novel methods to detect SDC-causing chips and improving best practices in SDC testing.

The version of this document: defines an SDC-causing computing system; and describes the information needed in a test input specification, test output specification, and part history specification. This version of the document also covers metrics on test effectiveness and details on test framework to help the academic community develop faster and better detection methods.

4.2. Out of Scope

This version of the document doesn't cover details on methodology and process for the research community to access faulty machines for experimentation. Access to faulty machines for test development will be coordinated with the participating companies separately meeting the individual requirements being met by the requesting entity.

5. Definitions

5.1 SDC-causing Computing system definition

A faulty machine is categorized as SDC-causing if we can identify a (bug-free) workload (e.g., a test created by SDC screening tools or an actual application workload) that produces incorrect results without any indication from the built-in error detection mechanisms.

6. Introduction to Information Exchange Specifications

Sections 7-8 describe a standardized information exchange to enable collaboration across academic institutions, hyperscalers as well as silicon vendors and tooling communities.

It is important that the communication around the progress and results have the same definitions and a standard format across academia and industry. To enable sharing information using the same format amongst each other, we propose an information exchange format. This section is the first attempt at enabling standardization around the information shared related to an SDC causing computing system as defined in the *Definitions* section.

Information Exchange Format:

This is written with the intent for use in communication with academic institutions. We anticipate that this will lead to new techniques for overcoming the SDC challenge. Examples of such techniques include new system-level tests (that may be provided to fleet quarantine pools and Data Center (fleet) owners can provide standard outputs for the submitted tests) and new scan-based tests that may be applied to designs supporting fleet wide scan (or may be studied using simulations).

A standardized format allows for iterations across academic teams and also allows us to compare effectiveness of various techniques. There would be limitations across quarantine pool populations between new machines flowing into the pool as well as machines flowing out due to decommissioning of servers within a fleet.

Sections 7-9 are a list of key fields that describe what needs to be provided with a new test, what the output report will contain after testing is completed, and useful information about identified defective parts. The screens or detection methods the academic community develops will be run on a pool of defective machines. The goal is to establish a library of parts and its characteristics in a database to enable ease of access with information sharing.

7. Test Input Specification

Key fields:

1. Tool configuration:

- a. **Required.** Indicates which tool or tool suite to use and desired command-line arguments. Optionally, this may include specific version numbers or even specific builds (e.g., compiler choice, static vs dynamic linking, etc.). In the case of a suite of tools or a tool-running tool, all indirect tools need to be specified too.
- b. *Reason for this field:* to provide the entity executing the testing the necessary information on what to run to ensure proper reproduction.
- c. *Suggested practice:* specific binaries (if acceptable) or source code to be built in a container.

2. Software environment selection:

- a. Indicates what other software must be present in order for the tool to run (usually dynamically-linked libraries), operating system version and any necessary OS-level tweaks, such as whether to run with administrator/root privileges, changes to /sys or /proc/sys (Linux) or in the registry (Windows), etc.
- b. *Reason for this field:* to clarify the necessary permissions the tool will need as well as to limit variance caused by different ancillary software versions.
- c. *Suggested practice:* virtual machine image, container, or container-building instructions (e.g., Dockerfile), plus host configuration.

3. Hardware environment selection:

- a. Indicates the target desired hardware (see output fields below for recording) as well as any ancillary hardware that must be present, such as accelerators, amount of main RAM, and whether access to said hardware must be bare metal, virtualized, or if it is unimportant. If relevant, firmware versions to be installed.
- b. *Reason for this field:* for the target device, to specify what the desired test is; for others, to ensure they do not affect the results or to verify whether they do.
- c. *Suggested practice:* node network identifiers, serial numbers, Best Known Configuration settings.

4. Desired run-time:

- a. Indicates the run-time to ensure data collection accumulates sufficient samples.

The input may be as vague or as detailed as necessary. For example, it is not necessary to specify a specific operating system version and runtime if those aren't known or suspected to influence the results.

8. Test Output Specification

Key fields:

1. Device type and device architecture:

- a. Indicate at a high level what kind of computing device it is. E.g., processor, GPU, AI hardware, etc.
- b. *Reason for this field:* to provide a higher level device aspect to the research community. It can also showcase that the SDC problem isn't confined to specific kinds of devices.

2. Technology Node:

- a. Include technology node information
- b. *Reason for this field:* to provide a higher level overview on the technology node at which the problem manifests. This can aid to show that the SDC problem isn't confined to specific technology nodes.

3. Tests applied during Vendor Manufacturing:

- a. Provide high-level information regarding the type of tests (workload) which were run to determine the component as SDC-causing. This is limited to the tests executed as part of manufacturing testing.
- b. *Reason for this field:* Many researchers do not have exposure or access into numerous aspects of testing parameters during manufacturing testing. To broaden the understanding of manufacturing tests which did not catch these defective chips will be important for innovative solutions. Moreover, for time $t > 0$ degradation, the burn-in/stress test information could be important.
 - i. *Scan tests:* coverage, test types, voltage, temp (at least qualitatively, hot/cold, high V, low V), frequency.
 - ii. *Functional/system-level tests:* Similar fields as above.
 - iii. *Burn-in and other stress tests*

4. Tests applied post-manufacturing:

- a. Provide high-level information regarding the type of tests (workload) which were run to determine the component as SDC-causing, these tests are limited to post manufacturing stages and in the datacenter. This would represent the tests or test suites that are run on machines to detect SDC-causing chips or machines in the datacenter as part of pre-deployment or *datacenter acceptance testing*. This field is to include all the **failing test names and test time per chip** to help with any correlation.
- b. *Reason for this info:* It will help a lot if the research community understands the type of tests which are failing and are effective in detecting SDCs. Providing high level data terms like system-level tests while failing to provide any insights would not yield research progress. Test times are important for the following reasons: (a) for test content optimization from manufacturing through in-field testing, (b) understanding various trade-offs and system-level adaptation, (c) test length plays a critical role in estimating coverage metrics.

5. Test environment:

- a. Provides an overview of the test environment and the related system level conditions. For example: were these tests run on bare-metal or VM/container based environments.
- b. Associated kernel and firmware versions while the tests were executed if they could be shared publicly. Any associated device configs which could affect the environment and could be shared would be included within this field.

6. Time to test failure:

- a. Represents the time it took for the test to fail for a particular failure. It could also represent aggregate metrics, which quantify average test failure time per architecture per test type. This represents the time from which a test was initiated on a machine to the time towards first detection of an SDC-causing failure.

7. Reproduction rate across test iterations (minimum 10 iterations):

- a. Represents the ability for the test to detect a failure across multiple iterations, a minimum of 10 is noted here as an example. However, based on the failure mode and nuance associated with the failure a different iteration denominator could be used (and explicitly stated) to arrive at a better characterization. The threshold could be subject to further discussions or modifications based on feedback and testing efficacy.

8. Error detection latency (if available):

- a. It is the time elapsed between creation of an error due to a defect and its detection by various tests (workloads). It is understandable that tools that are available today within the industry cannot accurately quantify such a vital metric – the key to understanding the origins of the SDC problem and eventually creating solutions.

This field has nuances associated as inherently there will be a delay between the true origin of defect and the time at which the defect is recognized. This field quantifies the risk of the delay in detection of SDC-causing defects within a chip.

- b. This has direct bearing on coverage, diagnosability (essential for test content coverage). It is also important for test times and detection efficiency.
- c. There are many ways of quantifying error detection latencies.
 - i. Hardware support (e.g., signature analyzers)
 - ii. Instrumentation of existing system-level tests with various granularities of checking.
 - iii. Fine-grained checking in hardware or in software
 - iv. Fault-tolerance techniques

9. High level failure info:

- a. Represents a high level failure information regarding the test failure. This could be information related to functional block or the nature of the failure i.e hard crash on test vs fail-continue on test and other aspects associated with silicon. Include specific information on the tools that aided in catching the defect.

10. Example log (if tool is open sourced):

- a. Example log would be shared if the tool is open sourced or any specific tools log that a vendor is willing to share. This standardizes test development and enables automation of log parsing.

9. Part History Specification

Once a defective part is identified, the following key fields describe useful characteristics about the part that may help test authors understand the sources of failure and how to create better tests to find them. Because these details can reveal some vendor-proprietary information (e.g. manufacturing processes, failure analysis methods, failure rates), some vendors may choose to provide them only under a Non-Disclosure Agreement (NDA).

11. Age at first failure:

Represents the age of the device at which the test failed for the first time and was confirmed defective. For testing within the quarantine pool, age may be a misleading factor. [Age here is measured from the manufacturing date]

12. $T = 0$ failure or $t > 0$ degradation

Age at first failure would not provide a complete answer on the origin and manifestation of the failure. For example, a $T=0$ failure might be detected after 1 year (new tests with coverage, same test detected later due to specific microarchitectural states, etc.).

Additional telemetry and understanding of coverage is required to truly characterize a failure as degradation at the chip level. At the test level, this can represent if more defects are detected with time in fleet and repeated testing for the same content (or) if the failure manifested and was uncovered because of a novel test path.

Reason: This information is important for understanding SDC mitigation strategies and testing costs within a large-scale fleet.

13. Additional Diagnosis details as available from vendors (assuming every chip passed manufacturing tests and failed system-level tests post-manufacturing)

- a. **13a** What kind of Failure Analysis was done and results
- b. **13b** Scan-based diagnosis information, if available
 - i. What scan test detected it together with test conditions?
 - ii. What were the fault diagnosis results?
 1. Timing-independent combinational defect
 2. Sequence-dependent (timing-independent) defect
 3. Timing-dependent

4. E.g., Stuck-at, bridging, open faults (how many fault candidates)

10. Example of Test Output and Part History

Below is an example to show what information is intended to be shared for test output and part history. The goal is to build a database of this information on pools of machines referred to as quarantine pools (or Q-pools) that contain both known-bad and suspected-bad machines. More information about how Quarantine pools will be managed will be forthcoming in a future revision of this specification.

Open Compute Project • Server Component Resilience

	Chip #		1	2
1	Device type/architecture		CPU	CPU
2	Tech node		7nm	14nm
3	Tests applied during manufacturing (Vendor to provide)	Scan tests (type, coverage, conditions) that caught the failure	N/A	N/A
		Functional tests (type, coverage, test conditions) that caught failure	Functional	Assembly + C (functional)
		Burn-in & other stress tests	N/A	N/A
4	Test applied post manufacturing together with test conditions		CPUcheck, 10 min	X87 test, 15 min
5	Test Environment: Tests run on bare-metal or VM/container based environments		Bare-Metal	Bare-Metal
6	Time to Test Failure 6a. Tool that triggered failures 6b. corresponding time		6a. Rainbow, 6b. 36 min	6a. X87 Test, 6b. 1 min
7	Reproduction rate of system-level tests and environment		100% @ Q-Pool	100% @ Q-Pool
8	Error detection latency		NA	NA
9	High-level failure info		Rainbow	X87 test, instruction specific failure
10	Example log		Attach the file	Attach the file
11	Age at first failure 11a. SLT tests during the machine life 11b. For Scan based test, include if its the same		11a. T=0 at QA, SLT - rainbow	11a. T=11 Mon
12	T = 0 failure or T>0 degradation (when fleet scan testing is enabled, this should be available)		T= 0 failure	T > 0 (Likely)
13	Additional Diagnosis details (as available from vendors)		-	Reproduced at vendor, Instruction failure at FPU stack to normal stack transfer (marginal timing scenario)

SLT = System-Level Test | VM = virtual machine | Q-pool = quarantine pool

11. Metrics

This section aims to define metrics that can be used to measure the SDC test quality and efficiency and to fairly compare different tests against each other.

11.1 Test Environment:

A test will be run on a pool of hardware containing systems with known defects, suspect defects as well as “known good” systems. A unified test framework may be used to submit the test, schedule the test on the available hardware and collect test results according to the proposed information exchange format.

Definitions:

	Definition
Hardware test pool	A collection of nodes with known defects, suspect nodes and “known good” nodes
Hardware sub-pool	Subset of the hardware test pool. Sub-pools may be created based on criteria such as: • Hardware failure state (known defect, suspect, known good) • Hardware type (CPU, GPU) and/or SKU • Defect class(es), for example: ALU errors, spinlocks. Comments: 1. Sub-pool size needs to be statistically representative. 2. The criteria described above are informative examples, the user may expand the list as needed or may choose which of the criteria to implement in their sub-pool classification.
Baseline test suite	Set of workloads/tests that expose known SDC-causing defects in a hardware test pool at a known rate. <u>Comments:</u> All new tests are going to be compared against the baseline to ensure a fair comparison. The baseline may be updated periodically (see pool snapshot version). Has to be run before and after collecting results with a new test/workload that is compared against the baseline.
Pool snapshot version	A collection of parameters fully describing a hardware test pool and baseline test suite, including test version number, node SKU information, firmware revisions, operating system revisions, etc. The pool snapshots (either the hardware test pool or the baseline test suite, or both) may be updated periodically to take into consideration: • new defect discovery

	• state of the art tests • introduction of new hardware • hardware degradation
Test uniqueness	Finds defects that have not been uncovered by any other test
Test coverage	Percentage of known defects found by a test
Test repeatability	The test generates the same test conditions (access patterns, stress levels, culprit-victim conditions, best effort to reproduce temperature, voltage, frequency, etc) upon multiple repeats in a system
Test reproducibility	Ability to reproduce the test results exactly upon multiple repeats.
Test cost	% system resources utilized for testing which can not be used for production activities
Time to fail (TTF)	Average time taken by a test to find a known defect
Test efficiency (TE)	A test's efficiency in detecting a known or unique defect. Inverse of the test's time to fail. Useful in characterizing tests because TEs are additive.
Defect	Results in a computational error and can include hardware manufacturing, latent and other in-field reliability mechanisms. Excludes functional bugs that affect 100% of a certain type of hardware.
Known defect	Characterized (known failure conditions and/or known root cause) and reproducible defect.
Suspect defect	Suspect device has encountered errors more than once. Failures are suspected to be associated with hardware defects but are not completely characterized.
Unique defect	A reproducible new defect observed in one or more servers in the test pool. It may be associated with the introduction of a new workload/test or new hardware.

11.2 Test Efficiency

A test may uncover one or more known defects. It is anticipated that each test will uncover hardware defects at different rates. The test efficiency metric compares different tests against each other based on efficiency.

$$TE = \frac{1}{TTF}$$

Where:

TE = test efficiency

TTF = time to failure in hours

A test may be applied to more than one sub-pool and may find different defects in each sub-pool. Assuming that the test being evaluated is running independently (not serialized) in each sub-pool, the average test efficiency across a pool will be calculated as the average of its test efficiencies across all the sub-pools which are part of that hardware test pool. For sub-pools where the test does not find any defects, the TTF is infinite and the TE is equal to 0.

$$TE_{pool} = \frac{\sum_{k=0}^n TE_{subpool}}{n}$$

Where:

TE_{subpool} = test efficiency for each subpool

n = number of subpools

Higher test efficiency numbers are better.

For unique defects or for cases where it is necessary emphasize the criticality of a defect over others, the user may choose to apply an additional weight factor when calculating the TE. The weight factor would emphasize the test efficiency associated with defects the user cares most about, but it introduces a level of subjectivity to the metric. For example: this factor could be chosen based on the impact the unique defect has in the datacenter. In some cases, if the new defect has negligible observed effects on the datacenter operations it can be weighed lower. If the unique defect causes significant downtime, increases the frequency of the checkpoint restarts and/or is found to propagate SDC to a significant amount of data prior to detection, the unique defect can be weighed higher than other known defects. The unique defect weight factor is going to be decided by the affected customer and can be applied to TE as follows:

$$TE_{pool} = \frac{\sum_{k=0}^n TE_{subpool} + weight_{factor} * TE_{unique}}{n+1}$$

The following examples show how the TE metric can be used to compare the efficiency of different tests aiming to uncover SDC-causing defects. The examples assume that the SDC-causing defects have been root caused, their fault models known and that the pool/sub-pool is of a statistically significant size relative to the amount of hardware of a certain type present in the datacenter. A sample of 1 failing piece of hardware is not excluded by the TE metric definition, but it does not meet the reproducibility criteria.

In the following examples, the term “test” refers to a complex test suite using multiple test patterns which can excite more than one defect. It is assumed that once a test finds a defect, it flags the defect and stops. The test finds different defects on different systems for example: on a system that has been screened for defect x (is known not to have defect x), the test could find another defect – defect y.

A targeted test which finds a single defect will but is run on multiple subpools should use the non-weighted TE formula (# TBA)

Example #1.0 - comparing two tests which find known defects in the same pool or sub-pool focusing on finding a defect with the highest efficiency

This example addresses the test efficiency associated with known defects and it assumes that the known defects have been root-caused (see known defect definition). The example focuses on a single pool/sub-pool (n=1)

	Test A TTF	Test B TTF	Test A TE	Test B TE
Defect x	2	0.5	0.5	2
Defect y	5	10	0.2	0.1
Total TE			0.7	2.1
Total test time	7	10.5		

Test B is very effective in finding defect x, making its test efficiency look better than test A overall. The total test time for test B is significantly longer than test A so the user will have to use their discretion when choosing between the two tests.

Use case example: Test B could be used to screen the pool efficiently for defect x. After the HW associated with defect x is quarantined either by creating a special sub-pool for it or through its removal from the pool, test A becomes the more efficient test for the remaining hardware.

Example #2 – accounting for unique defects found by a test in a pool or subpool
Example for a single pool/subpool.

	Test A TTF	Test B TTF	Test A TE	Test B TE
Defect x	0.25	0.5	4	2
Defect y	0.5	1	2	1
Unique defect	Infinite	0.5	0	2
Total TE (weight_factor =1) – unique defect is equally important			6	5
Total TE (weight_factor=2) – unique defect is more important			6	7=2+1+2*2
Total TE (weight_factor=0.5) – unique defect is less important			6	4 = 2+1+2*0.5

In some cases, the number of unique defects found or the size of the subpool (see example 3) may be used as a weight factor.

Note:

Using the size of the subpool may skew the test efficiency if there are large differences in the sizes of the subpools within a certain pool.

Test efficiency is a flexible metric which can be used at the experimenter's discretion to emphasize finding new defects or to focus on the timely detection of existing defects.

Example #3 – usage of TE when the SDC-causing defect is detected through a test timeout
A timeout is a valid method of detecting SDC-causing defects and many tests provision for timeouts if a desired response is not received from the hardware under test. For a timeout to be considered part of this example: the test should excite the SDC-causing defect and detect the timeout caused by the defect. If the timeout is not detected by the test itself, but by another HW or SW entity, then the experimenter needs to root-cause the defect to ensure that the timeout is indeed the result of an SDC-causing defect excited by the test.

TTF = duration of timeout

12. Test Framework and Test Flow

This section describes the available test frameworks researchers may use as well as the process for using the frameworks, requesting time on our machine pools, and analyzing results.

12.1 Pool access

Many Server Resiliency workstream members have gathered machines that are either known to or suspected to be affected by a hardware issue causing silent data corruptions. Those machines are in quarantine pools, isolated from regular workloads, but may be used to test effectiveness of tools in scanning for and detecting those defects. Researchers and partners may request time on those machines by emailing the workstream, at scr-feedback@ocproject.net. Information on what is suspected or known about those machines may also be requested by email.

The majority of nodes in those quarantine pools are not directly accessible by researchers. Instead, the researcher needs to discuss with the quarantine pool manager what workload is requested, on how many machines and of what type, and how to aggregate results. See the Test Input and Output sections above for information on how to specify that.

Please note that access is not guaranteed and will be decided on a case-by-case basis. Using the test frameworks described below is recommended, as they make the security review and integration with existing tooling easier.

12.2 Anatomy of a test

All tests share a common flow, which at high level can be summarized as follows:

1. Initialization: create execution plan, generate input data, calculate expected results, etc.

2. Execution: dispatch workloads to compute device units under test
 - a. Operate on input data and create candidate results, often comparing against expected output
 - b. If necessary, repeat multiple times to place unit under stress
3. Finalization: obtain results from the different compute device units to determine whether defects have been detected, produce logs and verdict
4. Clean-up: free any acquired resources

What a compute device unit is will depend on the device in question. For example, for a CPU, a compute unit is usually a logical processor (a core or hyperthread); for a GPU, it could be an execution unit such as tensor cores, media encoder/decoder engines, etc. For the common types of computing devices, the frameworks implement enumeration of the units available and dispatching to them, while ensuring exclusive access. Researchers are advised to reuse this functionality where available.

The simplest type of test is to execute a computation over and over again with the exact same inputs, comparing the output against previous executions and across different compute units, which is effectively a single-threaded workload. More complex tests could involve communication between homogeneous or even heterogeneous compute units, which add the variability of timing, data coherency, and signal transmission.

A test workload will often involve running one or more tests multiple times, with different input data.

12.3 Test frameworks

This section describes the different test frameworks that the workstream members have made open-source. They have different features, capabilities, and requirements.

The test frameworks are currently not compliant with this specification's input and output format, but work is in progress to achieve that. Please consult the framework websites for more current information.

12.3.1 Data Center GPU Manager

Link	https://github.com/nvidia/dcgm (Contribution documentation)
Vendor	NVIDIA
Compute devices supported	GPU
Programming language	Python
Operating system	Linux

License	Apache 2.0
---------	----------------------------

Summary

NVIDIA Data Center GPU Manager (DCGM) is a suite of tools for managing and monitoring NVIDIA datacenter GPUs in cluster environments. It includes active health monitoring, comprehensive diagnostics, system alerts and governance policies including power and clock management. It can be used standalone by infrastructure teams and easily integrates into cluster management tools, resource scheduling and monitoring products from NVIDIA partners.

Additional DCGM information can be found at: <https://developer.nvidia.com/dcgm> and <https://docs.nvidia.com/datacenter/dcgm/latest/>

Requirements

DCGM supports NVIDIA GPUsm it is recommended to use a Docker build image which is easier to build and provides a repeatable environment.

Root (privileged user) access required

Writing new tests

Information to be posted on the DCGM github.

12.3.2 OpenDCDiag

Link	https://github.com/opendcdiag/opendcdiag (Contribution documentation)
Vendor	Intel
Compute devices supported	CPU on-CPU-die accelerators
Programming language	C, C++, assembly
Operating systems	FreeBSD, Linux, Windows
License	Apache 2.0

Summary

OpenDCDiag is an open-source project designed to identify defects and bugs in CPUs. It consists of a set of tests built around a sophisticated CPU testing framework.

OpenDCDiag is primarily intended for, but not limited to, Data Center CPUs.

Requirements

Running as root and on non-virtualized environments is recommended but not required.

64-bit CPU (tested only ARM and x86 processors, both Intel and AMD)

Writing tests

OpenDCDiag is a C/C++ executable that must be compiled. The instructions for building it are available on the [OpenDCDiag project page](#).

New tests must be added to the build system and consist of three C/C++ functions, "init" (optional), "run" and "finish" (optional), which match the flow steps described above. Logging functions are also available for including extra information.

OpenDCDiag has a tutorial on writing tests:

https://github.com/opendcdiag/opendcdiag/blob/main/docs/writing_tests.md

OpenDCDiag provides existing, production-quality tests:

<https://github.com/opendcdiag/opendcdiag/tree/main/tests>

12.3.3 Open Field Health Check

Link	https://github.com/amd/Open-Field-Health-Check (Contribution documentation)
Vendor	AMD
Compute devices supported	Agnostic (enumeration & dispatching only for CPU)
Programming language	Framework is in Python, but tests may be in any language
Operating systems	Linux
License	MIT

Summary

OFHC run script is a test orchestration framework. You can give OFHC a configuration file with the tests to run and it'll run them in the desired sequence and log pass/fail information on each test. OFHC handles test execution on the desired cores of the CPU and checking for MCEs to make the process simpler for CPU test writers.

Requirements

Bare metal, non-virtualized environment
Root (privileged user) access required

Writing new tests

OFHC tests are separate executables that are launched and managed by the OFHC framework. It will therefore integrate with tests by many sources and may be used to combine multiple tests at the same time. Each test executable is a simple Linux

application that is executed on the requested compute units and produces a pass or fail result in the form of the result code from the executable.

Test configurations are specified in YAML or JSON. See [Configuration File Format](#) for more information.

13. References

- [1] <https://github.com/nvidia/dcgm> ([Contribution documentation](#))
- [2] <https://github.com/amd/Open-Field-Health-Check> ([Contribution documentation](#))
- [3] <https://github.com/openscdiag/openscdiag> ([Contribution documentation](#))

Appendix A - Checklist for IC approval of this Specification

Complete all the checklist items in the table with links to the section where it is described in this spec or an external document .

Item	Status or Details	Link to detailed explanation
Is this contribution entered into the OCP Contribution Portal?	Yes	If no, please state reason.
Was it approved in the OCP Contribution Portal?	Yes or No	If no, please state reason.
Is there a Supplier(s) that is building a product based on this Spec? (Supplier must be an OCP Solution Provider)	No	This is a 1.0 draft for feedback from the Community. No products will be built from this version.
Will Supplier(s) have the product available for GENERAL AVAILABILITY within 120 days?	No	