# Persistent Regions that Survive NVM Media Failure

One size does not fit all

Onkar Patil*
Department of Computer Science
North Carolina State University
opatil@ncsu.edu

Mesut Kuscu *
Technical University Munich
mesut.kuscu@hpe.com

Tuan Tran *
UC Santa Cruz
atran18@ucsc.edu

Charles Johnson
HPE Nonstop Research,
Palo Alto, CA 94304
charles.s.johnson@hpe.com

Joseph Tucek *
Amazon
Palo Alto, CA 94304
tucekj@amazon.com

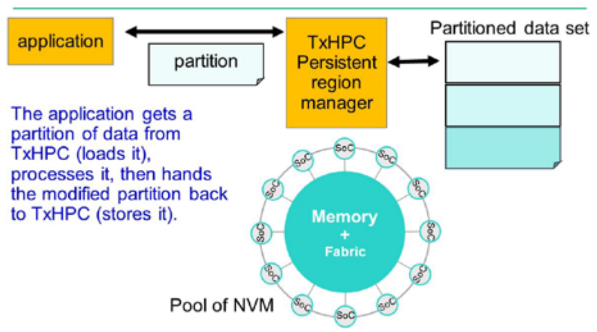
Harumi Kuno
Hewlett Packard Labs
Palo Alto, CA 94304
harumi.kuno@hpe.com

## ABSTRACT

We consider a pool of non-volatile memory shared by thousands of compute nodes. When a single computing node or memory unit fails, the remaining nodes and memory units may stay available, making it theoretically possible for long-running data processing jobs to survive media failure. In practice, however, customers face an all-or-nothing choice, and often choose to avoid the overhead of full hardware or software protection from media failure and instead limit the sizes of their data processing jobs to those that are likely to complete without failure. Our solution (TxHPC) enables staged data processing applications to survive media failure without requiring full-fledged hardware or software RAID. We have built an as-yet-unoptimized proof-of-concept of our solution and describe its design and implementation, as well as some preliminary performance results.

## 1. PROBLEM STATEMENT

Products and prototypes such as phase-change memory, Hysteresis RAM, and memristors are realizing today the long-awaited promise of persistent byte-addressable memory. However, like all other memories, these new devices are vulnerable to media failure. With traditional media such as hard drives, those who care about tolerance of media failure could turn to hardware RAID or a file system that provides software RAID. However, such solutions are all-or-nothing – they require the entire storage system to pay the overhead of a specific distributed RAID solution — and today's abundance of compute power makes that overhead increasingly prohibitive.

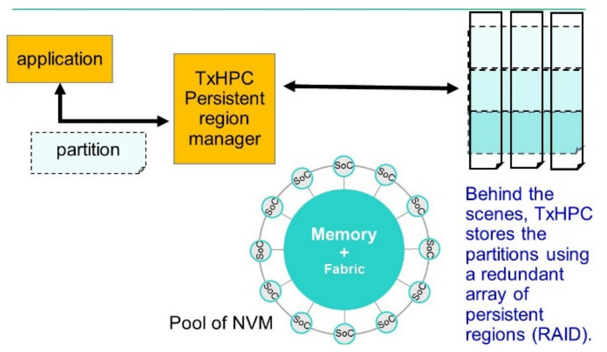


application
partition
TxHPC Persistent region manager
Partitioned data set
The application gets a partition of data from TxHPC (loads it), processes it, then hands the modified partition back to TxHPC (stores it).
Memory + Fabric
SoC
Pool of NVM

**Figure 1: From the application's perspective, it simply uses TxHPC to load and store data partitions.**

* This work was done while Joe, Onkar, Mesut, and Tuan were at Hewlett Packard Labs.

Since the entire state of a high performance computing system (HPC) is one big hotspot for update on each time-step, the traditional techniques used in reliability and resilience for systems like databases with a handful of hotspots and occasional queries that access mostly unmodified data, will break down quickly for HPC systems. Checkpointing, log writing and replication to make the state of the system recoverable would require transmitting the entire state of the system to safe store as the time-step is incremented in a typical simulation.



application
partition
TxHPC Persistent region manager
Memory + Fabric
SoC
Pool of NVM
Behind the scenes, TxHPC stores the partitions using a redundant array of persistent regions (RAID).

**Figure 2: From TxHPC's perspective, each data partition represents a stripe of RAID storage.**

## 2. APPROACH

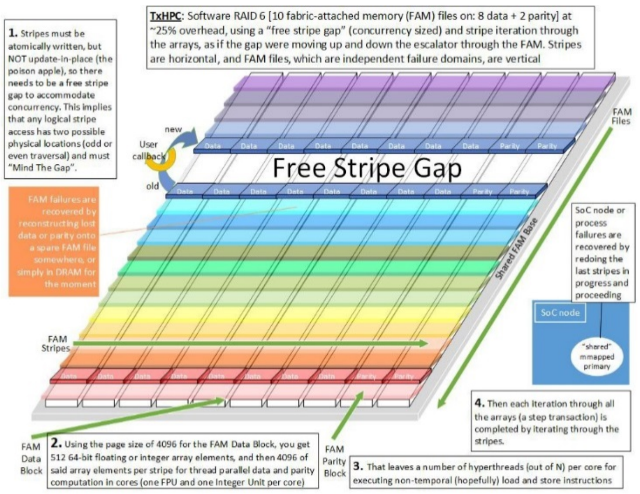
In general, TxHPC facilitates the development and execution of applications that run on many nodes with a pool of shared non-volatile memory. It allows large scale distributed applications that process data in stages to be coded in a manner completely transparent to the failure of both compute nodes and memory media nodes on the fabric, and also transparent to the tasks required to make the data safe and to allow the computation to continue across those failures.

Figure 1 sketches the workflow of how an application would use TxHPC from the application's perspective. To the application, its data set is stored in a persistent region of memory managed by TxHPC. TxHPC hands the application the data partition, the application processes the data, modifying its content, and then hands TxHPC new content for that partition to be stored in NVM.

Figure 2 sketches TxHPC's perspective. To TxHPC, each partition represents a stripe of RAID storage [1, 2]. Behind the scenes, TxHPC uses an array of persistent regions to store the data set redundantly. The memory fabric API is Gen-Z [3, 4].

The TxHPC persistent region manager is agnostic to the semantics and to the layout/organization/format of partition contents. It is also agnostic to the parallelization of compute within a stripe. In turn, the application is agnostic to the

**Figure 3: Example storage of a data set realized using 10 failure domains.**



TxHPC: Software RAID 6 [10 fabric-attached memory (FAM) files on: 8 data + 2 parity] at ~25% overhead, using a "free stripe gap" (concurrency sized) and stripe iteration through the arrays, as if the gap were moving up and down the escalator through the FAM. Stripes are horizontal, and FAM files, which are independent failure domains, are vertical

1. Stripes must be atomically written, but NOT update-in-place (the poison apple), so there needs to be a free stripe gap to accommodate concurrency. This implies that any logical stripe access has two possible physical locations (odd or even traversal) and must "Mind The Gap".

FAM failures are recovered by reconstructing lost data or parity onto a spare FAM file somewhere, or simply in DRAM for the moment

SoC node or process failures are recovered by redoing the last stripes in progress and proceeding

2. Using the page size of 4096 for the FAM Data Block, you get 512 64-bit floating or integer array elements, and then 4096 of said array elements per stripe for thread parallel data and parity computation in cores (one FPU and one Integer Unit per core)

3. That leaves a number of hyperthreads (out of N) per core for executing non-temporal (hopefully) load and store instructions

4. Then each iteration through all the arrays (a step transaction) is completed by iterating through the stripes.

actual physical data location of partitions and the persistence and crash recovery of the data within the grid/sub-grid or stripe/stripe-set.

The fabric attached memory (FAM) devices are divided data blocks of fixed sizes. Data is stored on these devices in stripes. Each stripe consists of the data blocks from each device that are at a specified common offset; each stripe contains a single data block from each memory device and 2 blocks from two memory devices reserved for storing parity. The parity is calculated for each stripe update.

Our persistent region manager is named Tx (Transactional) HPC because we read the stripe from persistent memory, process it in DRAM, and store the updated content back to another place in persistent memory in a single transaction, thereby not updating-in-place, for crash recoverability.

We keep stripe location(s) free before the application begins. Any update to the persistent memory will be written on these moving free stripe(s). Once the old stripe is written to completion onto the new stripe across the free stripe gap, the old stripe physical location becomes the leading edge of the free stripe gap. The Free Stripe Gap keeps moving in this fashion till it reaches the opposite end of the data region from where it started and only then back towards the end where it started. The movement of the Free Stripe Gap from one end to the other is called a time step. As the Free Stripe Gap can move only in two directions, all the stripes can only ever be in two locations. These locations are predictable based on the direction of Free Stripe Gap which can be easily maintained by a counter which is incremented at the end of every step. So the location of stripes can be either even or odd, step-wise. Figure 3 shows an example with 8 data blocks and 2 parity blocks per stripe.

## 3. EXPERIMENTAL EVALUATION

In Figure 4 we can see that execution times grow as the size of the dataset grows. The execution times of the stencil code algorithm with TxHPC and without any failure induced is 5x slower than the stencil code algorithm implemented without TxHPC. This overhead is due to the parity calculations for every stripe. That overhead is proportional to execution times of the stencil code algorithm for all dataset sizes. Also, the overhead increases when we induce failures. That increased overhead is due to the added cost of reconstructing two blocks of data per stripe. The execution time for recovering from two memory failures is 10x slower than the stencil code running without TxHPC.

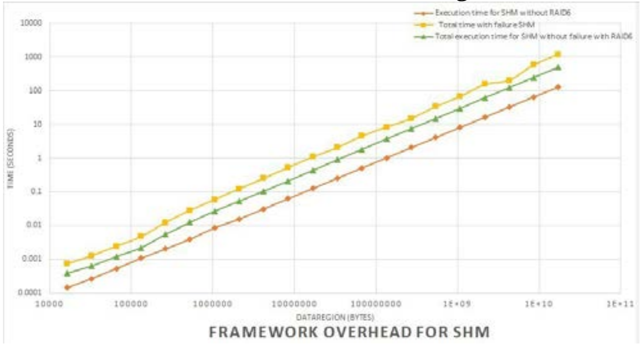


**Figure 4: Runtime overhead**

## 4. RELATED WORK

To our knowledge, although there are persistent memory programming approaches that provide data processing applications resilience against process failure or node failure, and although there are NVM storage systems that can protect against media failure, no other persistent region manager provides regions of persistent memory that can survive media failure.

The advent of byte-addressable persistent memory blurs the line between data management systems, programming languages, and file systems as programming language [5], storage system [6], and operating system [7] researchers all address the problem of how to ensure the consistency data stored in persistent regions in the face of failure.

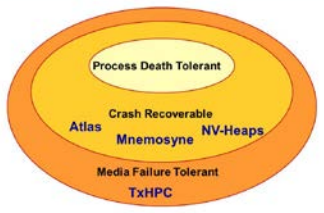


**Figure 5: To our knowledge, only TxHPC protects data in individual persistent regions from media failure.**

## 5. REFERENCES

[1] J. Plank, K. Greenan, and E. L. Miller. Screaming fast galois field arithmetic using intel simd extensions. In *Proceedings of the 11th Conference on File and Storage Systems (FAST 2013)*, Feb. 2013.

[2] J. S. Plank and K. M. Greenan. Jerasure: A library in c facilitating erasure coding for storage applications–version 2.0. Technical report, Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[3] Gen-Z Core Messaging Presentation. http://genzconsortium.org/wp-content/uploads/2016/05/Gen-Z-Consortium-Briefing-Deck_Final.pdf

[4] Gen-Z Introduction and Executive Summary http://genzconsortium.org/wp-content/uploads/2016/11/Gen-Z-Overview-V1.pdf

[5] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[6] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. *SIGPLAN Not.*, 46(3):105–118, Mar. 2011.

[7] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.*, 47(4):91–104, Mar. 2011.