A Comprehensive Approach to Memory Reliability through Replication

Dinesh Reddy Chinta, Tejas Bhagwat, Aadhithya Kannan, Derek Chiou

***Abstract*—Memory reliability requirement is not uniform across applications and hardware, demanding the need for flexible replication techniques. Using hardware-software techniques, we explored novel techniques for such Memory reliability requirements that are not uniform across applications and hardware, which demands flexible replication techniques. In this paper, we explored novel techniques for such replication across two elements in the memory subsystem hierarchy: caches and DRAM. For cache replication, we propose a cache-coherency protocol with a new SlaveModified (SM) state that is consistent with the actual modified state. Such a strategy handles cache capacity better by distributing the replication across multiple cores. This method also reduces performance costs associated with cache pollution. For memory replication, we proposed a page-mapping technique with an associated coherency protocol and a map-chaining strategy to support multiple replications. With the redundancy spread across nodes and resulting higher flexibility of correction capabilities, we analyzed the benefits associated with stronger detection codes by relaxing correction capabilities at the chip level by analytically computing the reliability associated with multiple replications. Finally, we discussed the possible performance gains and the associated latency, memory, power, and network overhead. We argued that latency and memory overhead could be traded based on the use case. Thus, we analyzed and proposed comprehensive techniques for the memory reliability problem.**

***Index Terms*—memory systems, DRAM, cache, reliability, coherence**

I. Introduction

Recent trends in the reduction of transistor sizes has led to increased memory failure rates. These memory failure rates have affected DRAM-based memories as well as SRAM-based (cache) memories. The increase in these failures have led to an increased demand for memory reliability. These errors are induced by alpha radiation, electromagnetic interference, and sometimes even temperature variations. Temperature variance induced errors, in specific, tend to bypass quality assurance testing. After the RAM is shipped to the customer and runs for several hours, it will heat up and only then will these memory errors be revealed.

These memory errors are unacceptable for certain use cases. For example, in Belgian 2003 national election, a bit flip in a voting machine caused a candidate to get 4096 extra votes. Arbitrary bit flips in memory can also derail security guarantees. Row hammer attacks attempt to induce bit flips in DRAM and have been shown to allow unprivileged attackers to gain the ability to edit arbitrary memory. Row hammer attacks have also been used to escape sandboxes. As a result of these revelations, the systems architecture community has looked into sacrificing a little performance to produce more reliable systems.

Traditionally, various error correcting codes (ECC) were implemented to protect against these failures. This project focuses on a new technique to improve memory reliability in the case of multiprocessor systems. The project utilizes data replication across different nodes for error correction. Distributing multiple copies of important data across different nodes in a multiprocessor system leads to increased reliability. The validity of data reads is checked using an error detection or correction code of choice. In the event that the error cannot be corrected, the data can still be recovered from a node carrying replica data. Additionally, maintaining strict coherence between these data replicas also allows for performance optimizations. In order to take advantage of these performance optimizations, we developed a modified cache coherency protocol to keep the data and its corresponding replicas coherent. Many of the current memory reliability solutions also treat memory indiscriminately. However, in real systems, not all memory is of equal priority. The user may want to replicate one region of memory several times, while other regions of memory do not need to be replicated at all. We call this feature dynamic replication and it is vital that our solution addresses this too.

We propose a combination of memory and cache reliability. A memory fault in DRAM or SRAM can cause data loss, so we felt that our solution needed to address both issues. For cache replication, we started with the standard MSI protocol and made some slight modifications. Primarily, every processor write fetched the cache line in a shared state into m adjacent cores. Cache writes were streamed to all the replicas so that they remain in sync. The details of keeping this modified protocol coherent were discussed in the *Coherent Cache Replication* section. For DRAM replication, we replicated the data to multiple physical addresses across nodes. A directory was maintained to keep track of this at page granularity. If a block of memory failed the ECC validation, then that data could be read from a duplicate node. This replication would also allow us to direct reads to multiple nodes to improve performance.

II. Motivation

Memory errors can occur in any module in the memory subsystem hierarchy – cells, banks, chips, ranks, memory controllers, and caches. We analyze each of the potential places where these errors could occur and identify two places—caches & DRAM—where existing solutions are inadequate. We then propose a novel technique to improve the resiliency.

Cell Errors: Rapid miniaturization trends indicate that smaller geometry cells are more susceptible to errors. This

has led to 16-bit ECC in DDR5 for 64-bit data. Furthermore, simple techniques like row/column sparing are provided to mitigate manufacturing faults. Not only is this leading to higher internal ECC capacity overhead, but it also impacts performance since more ECC bits require more complex decoders.

Chip Faults: Internal Circuit errors affecting multiple banks, rows, and cells have become a prominent reason for unpredictable errors in DRAMs. Resultantly, several variants of chip kill have been proposed to tackle this issue. A chip kill essentially maintains a redundant chip to store EEC that could correct up to one complete chip failure.

Channel Errors & Memory Controller Errors: Board-level circuit failures impacting multiple chips cause channel failures. Virtualized ECC techniques have been proposed to handle such channel errors through OS-architecture support. Several other studies show that there is a need to tackle these errors. DDR5 implements techniques to mitigate channel errors by maintaining address and data bus parity EEC [1]. More sophisticated commercial techniques involving redundancy like IBM's RAIM [9] (RAID-3 fashion), and Intel's Mirroring scheme (Raid-1 fashion) are available to tackle a complete channel failure, but these techniques have limitations associated with higher capacity overhead, limited flexibility, etc.

Cache Soft Error: Soft errors are becoming increasingly common, leading to effective techniques for increasing cache reliability. [2] uses an 8-entry fully-associative replica-cache to address the issue.

III. Understanding Memory Reliability

A. Reliability

Reliability refers to the ability of a system to consistently produce correct outputs up to some given time. Reliability is enhanced by features that help avoid, detect and repair hardware faults.

1) Memory Reliability: When a logical state of one or more bits is read differently from its last observed state, it is termed as memory error. Memory errors can be caused by electrical/magnetic interference, problems with hardware, or data corruption in the process of transferring the data. Memory errors can be classified as follows:

1) Soft errors: Randomly corrupted bits, but not permanent errors
2) Hard errors: Corrupt bits in a repeatable manner

A system without error correction and detection can have fatal consequences where a memory error can cause crashes or the usage of corrupted data.
Hence, systems use ECCs for the detection and correction of one or multi-bit errors.

2) Error Correction Codes (ECC) & useful terms: In ECC, the message is encoded by the addition of redundant information along with the data packet. The redundancy allows the receiver to detect a limited number of errors along with the correction of these messages without re-transmission.

1) **CRC**: In CRC, a short, fixed-length binary sequence is calculated for each block of data to be sent and it is appended to the data to form a "codeword." When the codeword is received or read, the check value is compared with the newly computed check value from the data. If the CRC values do not match, the data block contains an error.
2) **Parity**: This is a special case of 1-bit CRC, where a parity bit is added to detect single-bit errors in a block. This technique is effective when the probability of error is low; hence, the corner case of error correction could be made expensive to reduce the overhead. In DDR5, this has been employed for address and data parity.
3) **Double Symbol Detect (DSD)**: Hamming and Reed-Solomon codes add a higher number of redundant bits to improve resilience for detecting errors and correcting them. By maintaining special properties of the messages, these codes detect two errors and correct one single error. Techniques like Virtualized ECC [8] have used 8-bit symbol-based RS (18, 16, 8) code to implement SSC-DSD (Single Symbol Correct-Double Symbol Detect) properties.
4) **TSD**: SSC-DSD can be enhanced to detect multiple errors, as studied in [5]. In our analysis, we consider the impact of Triple Symbol Detect on reliability and show that the system reliability could be improved with TSD ECC codes.
5) **Detected Uncorrected Errors (DUE)**: ECC mechanisms detect/correct errors in memory subsystems. It is important to quantify system reliability based on the underlying EEC techniques. Detected Uncorrected Errors (DUE) quantifies the reliability in terms of the number of detected errors in a given interval of time. In our analysis, we estimate DUE for the proposed techniques and compare them with the existing techniques.
6) **Silent Data Corruptions (SDC)**: While DUE estimates the probability of a system ending up in a degradable state, it is also important to understand when the current techniques fail to detect an actual error occurring. Our analysis shows how this can be computed and compares it with existing techniques.

IV. Proposed Techniques

One of the important goals of our model is to be flexible with which addresses to replicate and how many times that replication should happen. We describe two ways of handling this memory flexibility.

1) **CPU State bit method**: Applications are marked as reliable, changing the state of the CPU when the process is loaded on the CPU. CPU State bit method: The private cache controllers transform the Load/Store instructions into their corresponding bus operations to distinguish replication operations from normal.
2) **Marking pages for replication**: A variant of the MMAP system call is provided to enable higher flexibility. The page-fault handler sets special state bits in the PTE, marking that the page is replicated. At runtime, the MMU intercepts the addresses marked for replication and generates special bus/protocol operations for those addresses.

V. Coherent Cache Replication

In addition to the standard ECC that is used in today's caches to mitigate soft errors, we explore a holistic approach of data replication in caches. Cache replication has been explored by [2] by maintaining a separate fully associative cache for the replicas near L1. This approach of only protecting L1 is limited. Indeed, L2 is much bigger than L1 and hence is more susceptible to errors.

To tackle errors in L2 caches, we propose a modified cache-coherence protocol through replication to achieve reliability in L2 caches. Besides, the higher capacity of L2 caches makes it more ideal for such replication. It has been well understood and studied that some portions of memory need more reliability than others. Unlike [2], where reliability is maintained for all the cache lines, our main theme of selective replication is supported using the proposed cache-coherency scheme.

The directory controller initializes some cache lines in a new state called **SM** (SlaveModified) to indicate that the cache line is the copy of a modified cache line in a different core's cache. The **SM** cache lines can't be evicted by normal replacement policies in order to preserve the replication property of the cache line. When the real modified line is evicted, the replica cache lines simply get invalidated.

We are making the cache reliable with the following modifications to the standard MSI cache coherence protocol. The following are the differences between the standard MSI protocol and proposed protocol:

1) Every processor RWTIM would fetch the cache line into #replica adjacent cores in an SM state. We discuss optimizations associated with already existing cache lines in shared state in the discussion section.
2) Each time the processor writes to the cache line, the SM copies in the caches are updated.

In a normal MSI protocol, the shared state is the latest copy of the data. Hence to distinguish between a true shared state when the memory has the latest data and a false shared state where the data is a replica of the latest copy of a cache line in modified state–and hence needs to be preserved unlike shared–we maintain one more variant of Modified state (say SM) in our proposed protocol.

Shared Bus Implementation

In this implementation, the snoop hardware checks if the RWITM operation on the bus corresponds to a replicated address. Then, if the current core is within the range of #node_id + #replicas / (total number of nodes), then add the current line into their cache in slave-modified (SM) state.

A Directory-based Implementation

In a directory based implementation, the directory controller sends the data to the adjacent nodes along with the requesting node. A similar snooping scheme as the bus based implementation adds the cache line the slave modified state.

A. Performance Impacts

For every write from the processor, the Slave Modified cache lines are updated with the modified value. However, we need to understand the implications of such updates on the expensiveness.

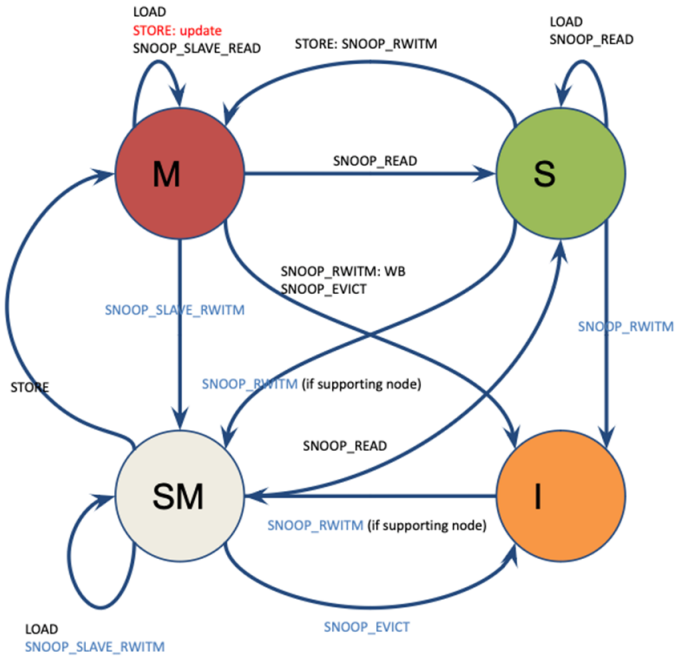

Fig. 1: State Diagram of Proposed Cache protocol for cache-replication

rep	Directory based implementation									Shared bus		
	Cache-tracked			Cache-tracked (circular)			Directory-tracked					
	S	L	O	S	L	O	S	I	O	S	L	O
x2	5	1	1	5	1	1	0	2	1	0	1	1
x3	10	1	2	5	2	1	0	2	2	0	1	1
x4	15	1	3	5	3	1	0	2	3	0	1	1

Fig. 2: Table of comparison between Memory, Latency and Network Bandwidth overhead for a 32 Chip Multi Processor. S = size (#bits), L = Latency(hops), O = bandwidth overhead (messages/operation). Cache-tracked represents the replica core information is kept inside caches; circular variant represents a linked-list type implementation; Directory-tracked represents directory keeps track of the replication cores; Shared bus represents snooping based protocol.

B. Discussion

1) Where is the replication information stored?: As shown in Figure 2, there are multiple ways of storing the replica cores' information.

In a directory based protocol, to send updates to the replicated nodes, the replica cache information must be stored

somewhere. If it is stored within the cache so that the cache can directly send the updates, the size of the cache becomes prohibitively large since the number of replications can vary. We may need to provision the space for the maximum supported replicability.

From figure 2, we can note that the proposed protocol, which involves updates, is most effective for shared bus type of protocol. It naturally obviates the need to replica information for updates to be effective.

2) What happens to existing Cache Lines in Shared state?: We leverage the existing shared copies in the other cores and only fetch the remaining number of replicas.

The implementation differs for shared bus and directory based implementation. In the shared bus implementation, when the cache-lines snoop a RWITM on the bus, they generate an UPGRADE busop in subsequent cycles to tell the memory controller about how many replicas are already satisfied. After the data arrives at the memory controller (memory access), based on the number of already existing shared copies, the controller responds with the corresponding #replicas information in the bus operation. The snooping hardware selects to transition to SlaveModified state based on the #replicas, current core_id, and the requesting core_id.

In a directory based implementation, the directory controller already knows which nodes are sharing the cache-line. Hence, it is much simpler to only send the data in SlaveModified state to the new replica cores and ask the current cores to upgrade to SlaveModified state.

To optimize this further, we can have some special counters for each core which represent the number of capacity misses from a cache. Such counters, when maintained in the directory-controller or memory-controller (for shared bus implementation), can be used to replicate the cache lines in less congested caches. This would alleviate the performance degradation associated with pollution.

This kind of optimization reduces the performance impacts associated with false evictions from the cores that actually require the data in shared state, and also the pollution associated with replicating the cache-line in an irrelevant core.

*3) Who is replicated and Where - **Only Modified State**:* Caches are built for speed. Though replication adds up to data reliability, Caches are very small. Hence, the replication cache lines have costs associated with capacity and could increase capacity misses if not modeled properly. This has been noted in [2] and argued that only modified data needs to be replicated.

When a shared cache line is corrupted, the shared cache line is simply discarded and re-fetched from memory, which has the latest copy of the data. However, for modified data though, since there is only one copy of the latest data, we must replicate the data and keep the replicas in sync with the modified data.

We have modeled our protocol to replicate only the modified data to keep the capacity and performance costs optimal (for the update messages add up to the network overhead).

In Write through caches, the modified cache lines are naturally modified in the higher level caches since the write propagates.

4) Who to evict?: Evicting a cache line in the SlaveModified state breaks our protocol and leaves the modified cache line with fewer replicas to guarantee reliability. Hence, we propose to give higher priority to SlaveModified cache lines compared to Shared lines. If all the cache lines are in the SlaveModified state, then we evict all (Modified/SlaveModified) cache lines to maintain reliability. This scheme is orthogonal to standard Cache-replacement policies since we add a new priority to SlaveModified cache lines. However, the rest of the cache lines still follow any standard replacement policy.

5) Capacity Overhead: [2] has very few entries in the replica cache. It did not completely address multiple replications. They maintained an 8-entry fully associative replica cache. In terms of power and energy, this extra fully associative cache appears minimal. However, our method does not require any new hardware to be added. Hence, we argue that we are more efficient in terms of silicon real estate and capacity.

6) Performance Overhead: The implementation also ensures that any data evicted from the replication cache gets evicted from the main cache. In other words, there would be contention if your working sets are bigger than eight cache entries, which is often the case for most array types of data structures. Besides, there is a double penalty of not only fetching the data but evicting it.

Our proposed protocol would have indirect implications on the capacity since we are polluting other caches with SlaveModified cache lines. Besides, we give higher priority to SlaveModified lines when deciding what to evict. On the workloads with working sets just large enough to fit in the cache, these replicated cache-lines degrade the performance due to increased capacity misses. However, to alleviate this problem, we argue that cache-miss counters can be used, as described previously, in the directory controllers to replicate to less congested caches.

While we don't represent normal cache lines (which do not need replicas), we argue that replica-mapped cache lines are differentiated from normal ones with some state-bits and the proposed protocol only has transitions for critical lines. This alleviates the update-overheads for normal cache lines.

VI. Memory Replication

A. Motivation

A much more useful thing is to maintain reliability in DRAM, where the probability of error is higher than in the caches made of SRAM.

Previous works focused on improving memory reliability at various levels – cells, chips, channels & memory controllers. Our work focuses on taking a holistic approach by augmenting reliability with coherent memory replication. This technique of page duplication is not new, our Problem Set 2.4 discusses providing coherency at cache line granularity by maintaining write permission per cache line of a page in the PTE. The design raises a page fault in the MMU, and the page fault handler transfers the data to the replicated nodes. Furthermore, Carrefour describes selective page replication and maintaining page-level coherency by maintaining write permission to only one page. Akin to those approaches, we maintain coherence

at the hardware level. Also, our primary focus is to improve reliability.

This idea has recently been explored by Dve to provide reliability through memory duplication. However, their approach did not address multiple replications. Also, their fully associative replica directory has capacity limitations. Besides, the fully associative directory size is prohibitively large (2K entries) to be efficiently implemented.

Unlike Dve, We solved the DRAM reliability problem by providing cache coherence between two different physical addresses, whereas Dve maintains replications under the hood locally, and coherence is maintained for the same physical address. Our main idea is to augment the existing coherence protocol by giving write permission to only one physical address. The coherence between the addresses is maintained through a new ReplicaModify state & Replica Map Tables (RMTs) – similar to Dve's RemoteModify state – to the existing MSI (Modified Shared Invalid) protocol.

B. Program semantics

For the sake of simplicity, we recommend a course-grained approach to maintain the replication at the page level. The PageTableEntries of the replicated pages is marked to be in the replicated state. When the page fault handler executes allocating a PTE since the addresses are marked for protection, the OS runs a FindGoodReplicationNode algorithm and determines the node where the replica page resides – we leave the exploration of such algorithms for our future work – it requests the replication node to allocate a replica page. If the node fails due to the unavailability of space, the requesting node fails and notifies the user about the unavailability of memory space.

If the replication node has enough space, it reserves a page for replication, marks the page as replicated, and responds back a success message to the requesting node along with the reserved page address. The requesting node then allocates a local page, marks it as replicated, adds the mapping to the RMT, and sends an ADD_RMT network command to the replication node along with the local physical page address. On the arrival of the ADD_RMT message, the replication node knows that the reserved page can be added to its local RMT table. In case of a FREE_RMT message from the requesting node indicating that it doesn't need the replication page anymore, the OS frees up the reserved page and unmarks it as a replicated page.

C. Directory structure

From the protocol, it is clear that we need to maintain a new state, ReplicaModified, for the protocol to function properly. The ReplicaModified state maintains that the current cache line can not be modified or read since the replica address has the most up-to-date copy of the cache line. As described in the protocol states, a network message is generated for the Replica cache line for LOAD and STORE operations. Then the Replica cache line, which is in the modified state, writes back the data to both the local and remote memory, and finally, sends acknowledgment messages for the current node to process with the LOAD/STORE operations.

Apart from the trivial extension associated with the new state and the busops, no extra bits are required for the replicated lines. In other words, we don't need to differentiate between replica and normal cache lines.

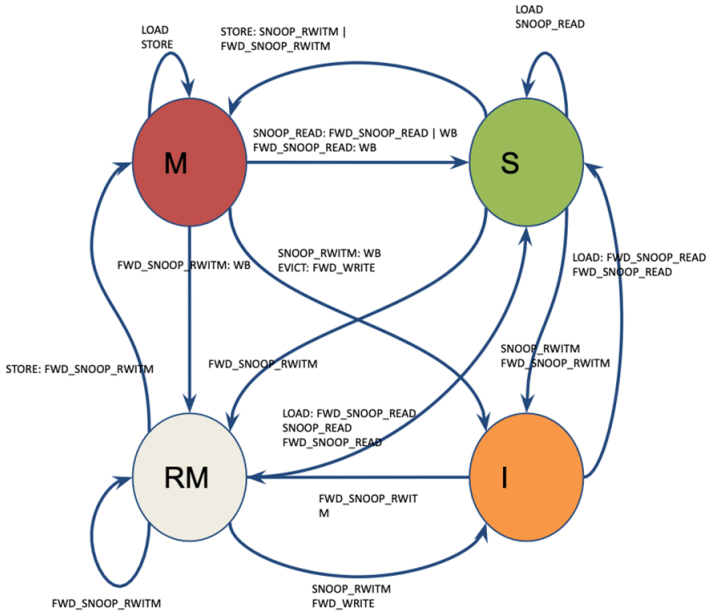

Fig. 3: Proposed Cache Coherence protocol for memory replication. RM = ReplicaModify state. All messages starting with FWD are the messages that are forwarded from other nodes.

D. Architecture and Implementation

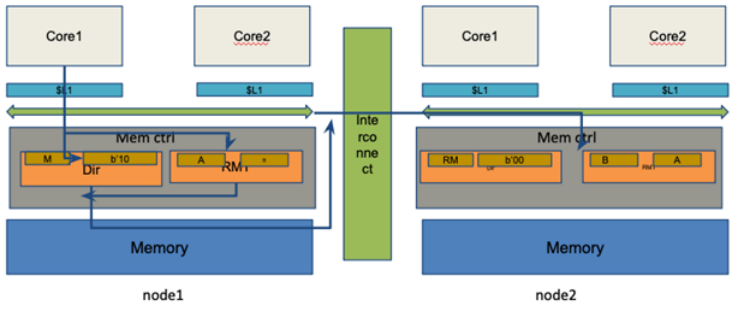

Fig. 4: System Architecture. Within a node coherence is maintained with a single address. Between two nodes, the dir-cntrl. uses a remapped address to send coherence/write-back messages.

E. Replica Map Table (RMT)

As described previously, this table maintains the mapping between two physical addresses that are coherent. For every network message that needs to be forwarded, the RMT is accessed, and the message is forwarded to the node responsible for that new address after the translation.

The structure of the table itself is similar to the page table that the OS maintains but instead of maintaining the

mapping between virtual to physical addresses, we maintain the mapping between two physical addresses.

1) How big is the RMT?: We maintain an entry to each of the page in the system. An alternative approach would be to get heuristics about the maximum possible size of the replications and provision those many entries in the RMT.

Aiming towards a generalized approach, we maintain an entry for each physical page the system supports. Besides, the complexity of searching this uniform RMT is reduced.

The RMT is filled during the page allocation stage. Deallocating the replica pages breaks the model. Hence, we block those pages from getting deallocated by local threads accessing the data. We initiate the chain of page-free commands when the process is killed.

directory1

Address indexed	State bits	Shared vector bits
A	RM	0x00 (remote modified)
...	...	...
A + page_size	S	0xff (all nodes shared)
...	...	...

directory2

Address indexed	State bits	Shared vector bits
B	M	0x01
...	...	...
B + 2 * page_size	S	0xff
...	...	...

RMT1

A	B
...	...
A + page_size	B + 2 * page_size
...	...

RMT2

B	A
...	...
B + 2 * page_size	A + page_size
...	...

Fig. 5: Example directory and RMT showing how the mappings work. Top left: directory of node1 where cache line at address A is in RM state & A + page_size address is in the Shared state. Top right: directory of node2 where cache line corresponding to address B and B + 2*page_size are in Modified and shares states correspondingly. For the corresponding snapshot of the directory states, the bottom two tables show the RMTs of node1 and nod2 where address A is mapped to B in RMT1 and B is mapped to A in RMT2.

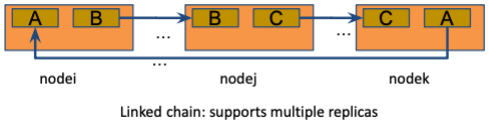

Fig. 6

F. OS support for Multiple replications

During the RMT initialization stage, the circular linked-list chain of replica maps should be generated correctly. To facilitate this, the requesting node also sends the yet-to-be-satisfied replica count information along with the total number of replicas. The FindGoodReplicationNode now returns the next replication node. Each time a replication node satisfies the request, it forwards a new request to the next node. After the replication count equals the requested number, a return chain of success messages starts until the requesting node is reached. In case a node does not have space for replication, FindGoodReplicationNode returns a new replica node. If the request cannot be satisfied, a FAILURE message is propagated, freeing up the reserved pages and notifying the user about the failure.

G. Discussion

1) Access Both Address From Same Node: By design, we do not support two physical address being used from same node. In other words, we maintain that the two concerned physical address belong to two nodes. Hence, we do not have to change internal hardware to support coherency between two physical address within a node.

2) Performance Gains due to replication: Carrefour and Dve showed the performance gains associated with replication. This would necessitate remapping the application code to the nearest replica. During the Process Initialization stage, the OS maps the shared data to the nearest physical address. This makes the application oblivious to the coherency maintained under the hood with a different physical address.

3) Overhead: As noted earlier, reliability increases exponentially with the number of replicas since the probability is inversely proportional to the number of replicas. We have the following challenges to implementing multiple replicas:

1) Memory overhead
2) Latency overhead
3) Network Congestion
4) Power and Energy

*a) **Memory Overhead**:* Three components add to storage overhead

1) The replica memory itself.
2) RMT in-memory data structure
3) Directory

Unlike Dve, which replicated all the memory twice, we selectively replicated only the mission-critical data multiple times. For a typical runtime, we argue that this might just be 1% overhead (may change depending on use cases). This efficient replication reduces the storage overhead by a negligible amount compared to other existing techniques. While we can venture into time-consuming compression techniques, this has a trivial latency overhead associated. Besides, as noted in Dve, approximately 50% of the memory in HPC systems is ideal. Hence, we claim the proposed method replicates the memory efficiently.

Another non-trivial storage overhead is with the RMTs. These are the in-memory data structures, as suggested in the Dv'e. Since these could get as big as the page tables, we argue that this is tolerable, provided that we save a lot by only replicating selective data.

Also, for storage, if we need to provision all the possible number of replications, it would waste a lot of space. We might maintain a fully associative cache for the replica map, but that won't be scalable. Thus, this linked list type of mapping also reduces the storage overhead. However, we would leave it to the user to optimize for latency if it is a primary concern.

*b) **Latency Overhead**:* In the most optimal case, to minimize the associated latency, each RMT entry should hold the mapping of multiple replicas locally, which is a challenge since we don't know how many mappings to provision for each RMT entry. Hence, we follow a linked list type of approach to optimize memory overhead. While this brings scalability, it adds to the latency since each message has to hop the replicas sequentially now.

As a worst-case analysis, we study the latency overhead in a circular linked list remapping where each invalidation and write-back become expensive for critical addresses. When RWITM comes for a shared cache line, the directory controller propagates the RWITM to the replica nodes, taking 2 * #replica hops for the acknowledgment to return. However, it should be noted that this is only required for the replicated address, and the normal address is not impacted.

Carrefour argued that interconnect network latency is gradually reducing; hence, we expect this latency to be tolerable in the future. For our design point, we optimize the latency by hiding the RMT access latency by parallelizing the access with the directory, as shown in the below figure.

*c) **Network Bandwidth Overhead**:* As explained with Dve we reduce the overall network traffic by servicing some of the read/write requests locally using the replicated memory. This is also discussed in Carrefour's paper, which showed that local replication reduces network bandwidth because the data travels less through the network.

But a nontrivial RMT translation overhead is associated with contacting other nodes for coherence. However, we argue that this is necessary. Moreover, this is only required for critical addresses. Furthermore, our linked-chain structure has the added advantage of distributing the network messages over time, alleviating burst patterns.

d) Power and Energy Overhead: We introduce new in-memory data structures, RMTs, it is useful to estimate the energy and power consumption of those elements. Since RMTs are in-memory data structures, static power consumption associated with the RMTs does not count. The dynamic power consumption is equivalent to the number of RMT access.

We proposed using a TLB-like structure to optimize the number of memory access associated with RMT access and the access latency. Since the structure only needs to hold the replicated mapping at page granularity, we argue that this structure is expected to be very small. Hence the power/energy consumption associated is negligible.

The third component is the extra bits added in PTEs for cache replication, extra bits for maintaining the cache coherency for both cache and memory replication. As a general trend, the number of bits associated with maintaining the cache coherence is 100. Hence, we argue that this additional power is also negligible.

e) Performance Impact on Unreplicated Memory: While the replication improves the reliability of some selected memory, it is also important to understand the performance implications on a general address that is not replicated. We retained the same states and state transitions as in the MSI protocol in our implementation. Hence, for normal operation, the latency is not impacted directly.

However, there are extra coherence messages generated during the transitions. But this overhead can be alleviated by trading off with extra state bits that would enable us not to generate extra messages for normal cache lines. An alternative approach is to trap the overhead messages at the RMT and discard them after finding that the cache line is not replicated.

VII. Analysis of Reliability for the Proposed Design

As discussed previously, DUE and SDC are useful metrics to quantify the reliability of a system. We follow similar methods to [4] for our analysis. Our model is based on the assumption of a uniform FIT rate of 66.1 [6]. Since the underlying devices are equipped with memory scrubbing (memory is refreshed periodically), our model computes errors if devices fail within a scrub interval.

Our memory configuration for the analysis is 32 single-rank EEC DIMMs, each DIMM with 9 chips. The extra redundant chip can be used in each rank to store ECC codes. We consider different ways of using the extra chip for storing ECC bits.

A. DUE

1) Chipkill: By definition, Chipkill recovers for single chip failures, and hence it fails when two chips fail simultaneously in a scrub interval.

$DUE = (\#ways\ of\ selecting\ two\ chips) * (first\ chip\ fail) * (second\ chip\ fail) * (\#DIMMS)$
$= (9 * 8/2) * (66.1) * (66.1/10^{-9}) * 32$
$= 5 * 10^{-3}$

2) Replication x2: When we replicate, DUE occurs when chips at the same location fail in two DIMMs, which is given by,

$DUE = (\#ways\ of\ selecting\ one\ chip) * (first\ chip\ fail) * (second\ chip\ fail) * (\#DIMMS)$
$= (9) * (66.1) * (66.1 * 10^{-9}) * 32$
$= 6.25 * 10^{-4}$

Note that here ECC detects only single symbol errors in the scrub interval. Thus the DUE is similar to Chipkill.

3) Replication (variable factor): When we replicate multiple times, the DUE reduces since all replicas must fail simultaneously. We can generalize the above formula to

$DUE = (\#ways\ of\ selecting\ a\ chip) * (first\ chip\ fail) * (\#DIMMS) * [(a\ chip\ fail)^{\#replicas}]$
$= (9) * (66.1) * 32 * (66.1 * 10^{-9})^{\#replicas}$

We observe that each replica adds an improvement of a factor of a chip failure rate.

4) Replications with Chipkill: This is an interesting configuration to analyze. Here, since Ckipkill's itself fails when two chips fail, the DUE's constant factor itself is two chip failures, given by,

$DUE = (\#ways\ of\ selecting\ a\ chip) * (first\ chip\ fail) * (\#DIMMS) * [replica\ failure^{\#replicas-1}]$
$= (9) * (66.1) * 32 * (66.1 * 10^{-9})^{\#replicas}$

We observe that a factor of 2 chip failures increases for each replica, i.e., $(66.1 * 10^{-9})^2 \approx 10^{-14}$

B. SDC

SDCs occur when the ECC itself fail to detect an error. Since this requires stricter conditions to be satisfied, we expect them to be lower than DUEs.

Unlike the DUE rate, the SDC rate can not be improved by adding replicas since each replica can fail independently. To improve this, Dve proposed retargeting the extra bits in the EEC DIMMs to improve detection capabilities by compromising on the correction capabilities. As studied in [7], TSD can be enabled with the extra bits.

We analyzed the improvements with such ECCs as a separate row in the table

1) Chipkill: If three or more chips fail in the same DIMM, then SDC occurs.

$SDC = (\#ways\ of\ selecting\ three\ chips) * (first\ fault) * (\#DIMMS) * (single\ chip\ fault)^2$

$\approx 10^{-10}$

We can observe that this is much lesser than DUE.

2) Replication x2: With DSD, a three-chip failure leads to SDC, approximately the same as Chipkill.

With TSD, a four-chip failure leads to SDC, which is by a factor of single-chip failure rate.

$SDC = (\#waysofselectingfourchips) * (firstfault) * (\#DIMMS) * (singlechipfault)^3$

$\approx 10^{-16}$

Replication	Scheme	DUE		SDC	
		Rate	Impr.	Rate	Impr.
	Chipkill	5 * 10 ^ -3	-	3.1 * 10 ^ -10	-
	IBM RAIM	1.5 x 10 ^ -14 [dve]	-	4.0 x 10 ^ -10 [dve]	-
X2 [dve]	DSD	2.5 * 10 ^ -3	4x	6.3 x 10 ^ -10	0.49x
	TSD	2.5 * 10 ^ -3	4x	**2.5 x 10 ^ -16**	**~ 10 ^ 7**
	Chipkill	**8.7 x 10 ^ -17**	**10 ^ 14x**	6.3 x 10 ^ -10	0.63
x3	DSD	2.5 x 10 ^ -10	10 ^ 7x	~ 10 ^ -10	-
	TSD	2.5 x 10 ^ -10	10 ^ 7x	~ 10 ^ -16	-
	Chipkill	**~ 10 ^ (-31)**	**10 ^ 28**	~ 10 ^ -10	-
x4	DSD	~10^-17	10^14	Same as x2	-
	TSD	~10^-17	10^14	Same as x2	-

Fig. 7: Reliability Comparison between Chipkill, IBM RAIM, Dve, Multiple Replications in terms of DUE and SDC

C. Discussion

The table in Fig.7 shows various DUE and SDC rates for different techniques that we studied with our baseline as Chipkill. Since our techniques have broader correction capabilities, we can detect errors more efficiently with TSD. Furthermore, since DUE scales exponentially with several replicas, we achieve an arbitrarily high DUE rate by increasing the number of replicas.

REFERENCES

[1] A. Patil, V. Nagarajan, R. Balasubramonian and N. Oswald, "Dvé: Improving DRAM Reliability and Performance On-Demand via Coherent Replication," 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021, pp. 526-539, doi: 10.1109/ISCA52012.2021.00048.

[2] W. Zhang, "Replication cache: a small fully associative cache to improve data cache reliability," in IEEE Transactions on Computers, vol. 54, no. 12, pp. 1547-1555, Dec. 2005, doi: 10.1109/TC.2005.202.

[3] R. Fernandez-Pascual, J. M. Garcia, M. E. Acacio and J. Duato, "A fault-tolerant directory-based cache coherence protocol for CMP architectures," 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), 2008, pp. 267-276, doi: 10.1109/DSN.2008.4630095.

[4] M. Taassori et al., "Compact Leakage-Free Support for Integrity and Reliability," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 735-748, doi: 10.1109/ISCA45697.2020.00066.

[5] Yeleswarapu R, Somani AK. 2021. Addressing multiple bit/symbol errors in DRAM subsystem. PeerJ Computer Science 7:e359

[6] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1-11, doi: 10.1109/SC.2012.13.

[7] X. Jian, H. Duwe, J. Sartori, V. Sridharan and R. Kumar, "Low-power, low-storage-overhead chipkill correct via multi-line error correction," SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1-12, doi: 10.1145/2503210.2503243.

[8] D. H. Yoon and M. Erez, "Virtualized ECC: Flexible Reliability in Main Memory," in IEEE Micro, vol. 31, no. 1, pp. 11-19, Jan.-Feb. 2011, doi: 10.1109/MM.2010.103.

[9] P. J. Meaney, L. A. Lastras-Montano, V. K. Papazova, E. Stephens, J. S. Johnson, L. C. Alves, J. A. O'Connor, and W. J. Clarke, "Ibm zenterprise redundant array of independent memory subsystem," IBM Journal of Research and Development, vol. 56, no. 1.2, Jan 2012.