Checkpointing Exascale Memory Systems with Existing Memory Technologies

Nilmini Abeyratne, Hsing-Min Chen*, Byoungchan Oh,
Ronald Dreslinski, Chaitali Chakrabarti*, and Trevor Mudge
University of Michigan, Ann Arbor, Michigan 48109
*Arizona State University, Tempe, Arizona 85281
{sabeyrat, bcoh, rdreslin, tnm}@umich.edu, *{hchen136, chaitali}@asu.edu

ABSTRACT

Building exascale supercomputers requires resilience to failing components such as processor, memory, storage, and network devices. Checkpoint/restart is a key ingredient in attaining resilience, but providing fast and reliable checkpointing is becoming more challenging as the amount of data to checkpoint and the number of components that can fail increase in exascale systems. To improve the speed of checkpointing, emerging non-volatile memory (phase change, magnetic, resistive RAM) have been proposed. However, using unproven memories to create checkpoints will only increase the design risk for exascale memory systems. In this paper, we show that exascale systems with hundreds of petabytes of memory can be constructed with commodity DRAM and SSD flash memory and that newer non-volatile memory are unnecessary, at least for the next generation.

The challenge when using commodity parts is providing fast and reliable checkpointing to protect against system failures. A straightforward solution of checkpointing to local flash-based SSD devices will not work because they are endurance and performance limited. We present a checkpointing solution that employs a combination of DRAM and SSD devices. A Checkpoint Location Controller (CLC) is implemented to monitor the endurance of the SSD and the performance loss of the application and to decide dynamically whether to checkpoint to the DRAM or the SSD.

The CLC improves both SSD endurance and application slowdown; but the checkpoints in DRAM are exposed to device failures. To design a reliable exascale memory, we protect the data with a low latency ECC that can correct all errors due to bit/pin/column/word faults and also detect errors due to chip failures, and we protect the checkpoint with a Chipkill-Correct level ECC that allows reliable checkpointing to the DRAM.

Using our system, the SSD lifetime increases by 2×—from 3 years to 6.3 years. Furthermore, the CLC reduces the average checkpointing overhead by nearly 10× (47% from a 420% slowdown), compared to when the application always checkpointed to the SSD.

MEMSYS 2016 October 3–6, 2016, Washington, DC, USA
© 2016 ACM. ISBN 978-1-4503-4305-3...$15.00
DOI: http://dx.doi.org/10.1145/2989081.2989121

CCS Concepts

•**Hardware → Fault tolerance;** Dynamic memory; Non-volatile memory; •**Software and its engineering → Checkpoint / restart;**

Keywords

fault tolerance; checkpoint/restart; ECC; exascale

1. INTRODUCTION

Aggregate failure rates of millions of components result in frequent failures in exascale supercomputers. In particular, exascale systems are projected to have memory systems as large as 100 petabytes—that is 100× larger than the supercomputer Titan's 1 petabyte memory system. The millions of memory devices that make up these memory systems contribute significantly to failures [1]. Overcoming these failures requires a fast and reliable checkpoint/restart framework.

Checkpointing—periodically saving a snapshot of memory to stable storage—is a useful practice to rollback the application to a point before failure, without restarting from the very beginning. Exascale systems rely heavily on checkpoints to recover from many types of failures including hardware failures, software failures, environmental problems, and even human errors [2]. Usually, checkpoints are made to a non-volatile storage such as a hard disk, but increasingly, solid-state drives (SSDs) are replacing hard disks because they provide higher read/write bandwidth, lower power consumption, and better durability [3]. The question becomes whether SSDs are sufficient for storing checkpoints or if we should wait for emerging memory technologies.

The biggest disadvantage of NAND-flash SSDs is its lower endurance, which is on the order of 10^4-10^5 program/erase cycles. SSD manufacturers employ various tricks such as DRAM buffers and sophisticated wear-leveling to extend lifetime. Currently, SSDs on the market are guaranteed a lifetime of 3-5 years with a cap on the total number of terabytes that can be written [4]. Nevertheless, writing gigabyte-sized checkpoints several times a day to the SSD can take a toll on its endurance.

Many have suggested using emerging non-volatile memory technologies such as phase change memory, memristors, and STT-RAM for checkpointing, often touting their superior read and write speeds and higher endurance [5, 6, 7]. While we do not disagree with these studies, emerging technologies must overcome many undeveloped steps between a successful prototype and volume production. It is difficult to guess when, or if ever, emerging technologies will be ready for the first round of exascale supercomputers. The U.S. Department of Energy's Exascale Computing Initiative plans to deploy exascale computing platforms by 2023 [8]. Designs for 2023 systems will have to be finalized 3-4 years prior, similar to plans for Summit (2018) and Aurora (2018-2019) supercomputers that were completed by 2015. At some point, system designers will have to reason about reliable, off-the-shelf components that will be available in the next 3 years. We show that existing non-volatile storage options that are proven less risky due their maturity and low cost are sufficient for the near future, if used correctly.

When using SSD flash memory for checkpointing, reducing the checkpoint size or frequency remain the most effective ways to stretch its lifetime. To this end, we propose a system that selectively checkpoints to a DRAM in order to reduce the number of writes to the SSD thereby lengthening its useful lifetime. To accomplish this task, we implement a Checkpoint Location Controller (CLC) that i) estimates SSD lifetime, ii) estimates application's performance loss, and iii) monitors checkpoint size. The CLC detects checkpointing frequencies that lead to SSD lifetime falling under the typical manufacturer's guarantee of 5 years, and reduces these frequencies by redirecting some checkpoints to the DRAM. We believe this is the first work to consider the lifetime of the SSD while writing checkpoints to it; previous work [9] that also used SSD ignored its endurance.

DRAM is prone to transient errors and checkpoints corrupted by them cannot be used for recovery. Then, a key feature to enable our technique to write fewer checkpoints to the SSD is to have a strong error correcting code (ECC) that can protect the checkpoints in DRAM. For that reason, we propose a dual mode ECC memory system that protects regular application data with a normal ECC algorithm and checkpoint data with a strong ECC algorithm. The normal ECC, which is on the critical path of memory accesses, is an RS(36,32) code that has small decoding latency to correct or detect errors. It can correct all errors due to a bit/pin/column/word failure and detect all errors due to a chip failure. The strong ECC is a two-layer RS(19,16) code that provides Chipkill-Correct level reliability without modifications to the DRAM devices. If an unrecoverable error corrupts the DRAM checkpoint, then the application will restart from the checkpoint in the SSD. The resultant capability to write reliable checkpoints to memory relieves the burden on the SSD, in turn lengthening its lifetime. More importantly, the combined DRAM-SSD checkpointing solution makes it possible to design an exascale memory system without relying on unproven emerging memory technologies.

In summary, we make the following contributions:

- **A low-risk exascale memory system.** We use mature technology in commodity DRAMs and SSDs to create a low design-risk checkpointing solution and prove that system designers do not have to wait until newer non-volatile memory technologies are ready.
- **Hybrid DRAM-SSD checkpointing**. Our local checkpointing solution is a hybrid mechanism that uses both DRAM and SSD flash memory to achieve speed and reliability (Section 3).
- **SSD-lifetime-aware checkpoint controller.** We design an intelligent Checkpoint Location Controller (CLC) that decides when to checkpoint to the SSD considering its endurance decay and performance degradation (Section 3.3).
- **Dual-ECC memory.** We propose a dual mode ECC memory that has a normal ECC mode to protect regular application data and a strong ECC mode to protect the DRAM checkpoint. ECC-protected checkpoints ensure error-free restarts at recovery (Section 4).

Our results from microbenchmark simulations averaged across various checkpoint sizes indicate that the CLC is able to increase SSD lifetime by 2×—from 3 years to 6.3 years—exceeding the guaranteed lifetime of 5 years [4]. Furthermore, the performance estimation feature in the CLC that monitors application slowdown is able to reduce the checkpoint overhead to a 47% (on average) slowdown, compared to a 420% slowdown when the application always checkpointed to the SSD—nearly a 10× savings.

2. MOTIVATION

Local checkpoints to local storage (DRAM or SSD) have stemmed from a need to avoid the slowdown resulting from transferring checkpoints to the remote parallel file system (PFS) over limited-capacity I/O channels. It is difficult to decide on the best local storage because each has their advantages and disadvantages. On one hand, DRAM is fast (50ns [10]) but loses the checkpoint after a reboot. Furthermore, limited DRAM capacity not only limits the size of the largest checkpoint that can be made but also limits the amount of usable memory for applications.

On the other hand, SSDs are reliable and capacious but slow and have low endurance (10^5 program/erase cycles). To illustrate the speed difference between *ramdisk*—a virtual disk created in DRAM to write checkpoint files—and the SSD, we measured the total runtime of a microbenchmark (details provided in Section 5.1) under three naïve implementations i) no checkpointing, ii) checkpointing to ramdisk only, and iii) checkpointing to SSD only. For this simulation, we assumed that both ramdisk and the SSD had unlimited checkpoint storage. As can be seen in Figure 1, writing the checkpoint to ramdisk incurs a small 14% slowdown, but checkpointing to the SSD incurs a considerable 4.6× slowdown averaged across all the checkpoint sizes.

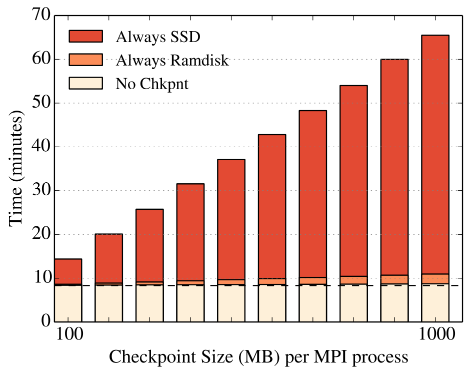


Figure 1: Microbenchmark runtime results with various checkpoint sizes demonstrate that always checkpointing to the SSD incurs significant overhead. Baseline runtime = 8.3 minutes.

The key insight gained from our experiment is that even when checkpointing only to the SSD, files still occupy memory space because they are first allocated in the memory's page cache. Files in the page cache are not necessarily flushed to the storage device when the file is closed because the OS delays the write process to hide I/O latency. On the other hand, explicitly flushing the page cache every time incurs overhead because the slow write delay to flash becomes fully transparent. If we try to use the OS's method of hiding latency (implicitly writing to the memory), then the checkpoint is sitting vulnerable in the memory and there is no guarantee when it will be persisted to the SSD. Therefore, a simpler and better approach is to explicitly write to both—to write a select few checkpoints to the SSD and always flush them and balance out the performance loss by writing the remaining checkpoints to the ramdisk.

The hybrid solution merges the benefits of both DRAM and SSD: namely, speed and reliability. Furthermore, checkpointing to the DRAM helps to reduce SSD wearout. The shortcomings of our solution is that it limits the available memory for applications and increases the memory pressure (i.e. ratio of active memory pages) due to active checkpoints residing in memory. The pros and cons of the proposed technique are listed in Table 1.

The checkpoints in ramdisk are exposed to DRAM failures, but ECC algorithms exists that are capable of protecting against most memory failures—except for a power outage. The stronger the ECC, the more time and power that it takes to decode data. A second key insight into our idea is that it is possible to use stronger ECC algorithms for checkpoints because decoding them is not on the critical path of normal application execution.

Table 1: The pros and cons of the proposed technique compared to DRAM-only or SSD-only checkpointing. The memory occupancy is marked as "Med" because the CLC can detect and send large checkpoints always to the SSD.

	DRAM only	SSD only	Proposed DRAM+SSD
Checkpoint Speed	Hi	Lo	Hi
Recovery Speed	Hi	Lo	Hi
Transient error protection	Lo	Hi	Hi
Available memory for apps.	Lo	Hi	Med
Memory pressure	Hi	Lo	Med
Non-volatile; persists reboots	N	Y	Y
Good SSD endurance	–	N	Y

Alternative memory technologies such as phase-change, magnetic, resistive RAM, and 3D XPoint holds promise because they are almost as fast as DRAM (10-300ns [10]), yet also as reliable as storage. However, these technologies are not yet as dense or cost-efficient as flash. Although Intel's 3D XPoint is expected to cost half of DRAM [11], recent innovations in 3D NAND-flash such as stacking 48 layers [12] will only cheapen flash. Furthermore, unlike emerging technologies, flash devices have well understood failure patterns and strong ECC codes to protect them [13]. Commercial availability and maturity of both DRAM and NAND-flash prove them a low-risk option for at least the first round of exascale systems. Should emerging technologies become better than flash, they can easily be integrated into our hybrid system and achieve even better performance.

In the remainder of the paper, we address two questions: 1) how to decide when to checkpoint to the DRAM or the SSD? and 2) how to design a strong ECC algorithm to protect the checkpoints without interference to non-checkpoint-data memory accesses? To answer the first question, we implement the CLC in Section 3.3 that is aware of the endurance limits of the local SSD device and the performance degradation from writing to it. To answer the second question, we introduce a dual-mode ECC design in Section 4 that can be dynamically encode data in either normal ECC or strong ECC depending on whether the data is normal application data or checkpoint data.

3. HYBRID DRAM-SSD CHECKPOINTING

An overview of the hybrid solution is presented in Figure 2. In our system, all compute nodes contain main memory consisting of x4 ECC-DRAM devices and one SSD flash memory device. In comparison to existing local checkpointing solutions which write to only the ramdisk or only the SSD [14, 15], the hybrid system writes to both. It can exist within a hierarchical framework where global checkpoints are still written to the remote PFS. Note that this system differs from *double checkpointing* in other work [14, 16] that write identical checkpoints to two platforms in "buddy" nodes. Double checkpointing wastes memory space. In contrast, the hybrid system writes only one checkpoint to one platform in a given checkpoint interval as illustrated in Figure 3. Although not implemented in this paper, a possible optimization is to implement the hybrid system on top of a buddy system, where either the ramdisk or SSD checkpoint is saved in the buddy's ramdisk or SSD, respectively.

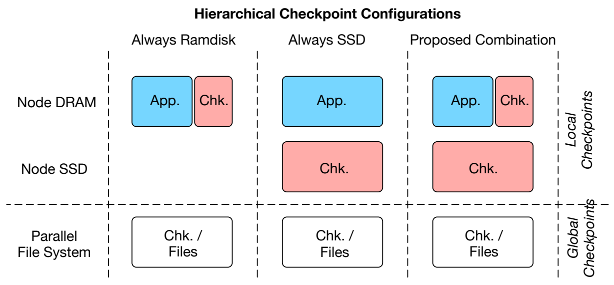


Figure 2: The proposed idea utilizes both commodity DRAM and commodity SSD for checkpoints.

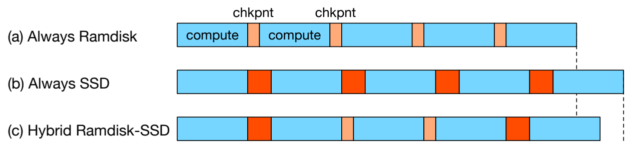


Figure 3: In the hybrid system (c), the CLC intelligently selects which checkpoints are to be written to the SSD considering endurance, performance, and checkpoint size.

3.1 Checkpointing to the Ramdisk

Checkpoints to memory are written outside of the application's address space to ensure its persistence after the application crashes or ends. This can be achieved by writing checkpoints to the *ramdisk*. There are two types of ramdisk file systems: *ramfs* and *tmpfs*. The main difference between them is that ramfs cannot be limited in size—i.e. it will keep growing until the system runs out of memory—whereas tmpfs will start swapping to disk once the specified size limit is full. We use tmpfs and enforce a size limit that ensures checkpoint memory does not encroach upon the application's memory.

3.1.1 Memory Requirement

In-memory checkpointing to DRAM requires prudent management of memory resources. Out of the available memory on each server node, a certain quantity is set aside for checkpointing by mounting a ramdisk into the memory space. The user should consider the memory requirement for both the application and the checkpoint. For example, 4GB out of a 24GB system can be set aside for checkpoints, leaving only 20GB for the application. The high performance application running on the node can be adjusted for the smaller memory size by setting a smaller problem size per MPI process, or by running fewer MPI processes on the node.

3.2 Checkpointing to the SSD

Writing checkpoints to SSDs have a history in "burst buffers" [9] and diskless checkpoints [17]. Several supercomputers that will be built between 2016-2019 such as Cori, Summit, and Aurora, all plan to include persistent memory in each compute node in the form of an SSD [18].

Our system uses **application-level checkpointing** in which the programmer carefully selects the data to be saved such that the program can be successfully restarted with that data. The data is written out in the format of a file, and storing and retrieving the file is handled by the file system on the SSD. Usually, when writing a file to any storage device, it is first temporarily allocated in the memory then flushed to the device later. To ensure the file has persisted to the SSD, the Linux *fsync()* operation must be called after each checkpoint. Otherwise, there can be no guarantee the file is recoverable after a crash and reboot.

3.3 Checkpoint Location Controller (CLC)

The CLC writes checkpoints to the ramdisk or to the SSD by setting the file path to point to either the ramdisk or the SSD. The decision is made just before the application starts writing each checkpoint. The CLC can maximize the lifetime of the SSD (Section 3.3.1), and/or minimize the performance loss of the application (Section 3.3.2). It can also take into account the size of the checkpoint (Section 3.3.3). An overview is in Figure 4. Section 3.3.4 shows how all three metrics are combined into one algorithm used by the CLC.

3.3.1 Lifetime Estimation

The endurance of an SSD is described by bytes written (e.g. TBW–terabytes written or PBW–petabytes written), which is the total amount of writes that it can withstand without wearing out. To obtain an example for the lifetime of a real device, we chose the Intel DC S3700 SSD in 800 GB as a reference [4]. Intel's "DC" data-center SSD's are some of their highest endurance SSDs suitable for high performance computing. The S3700 reported an endurance rating of 14.6 PBW [4].

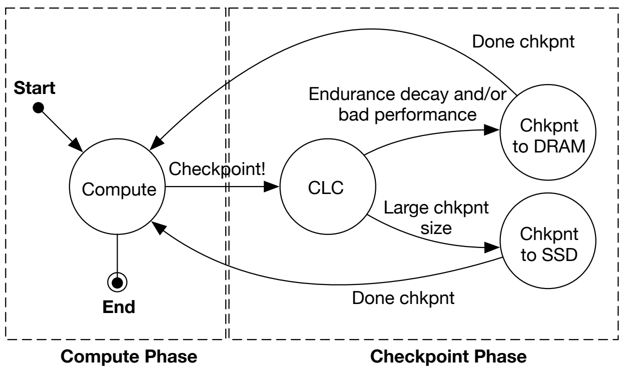


Figure 4: This state machine representing application execution shows how in the checkpoint phase the CLC dynamically decides the checkpoint location on each iteration.

To measure endurance decay, the CLC calculates an 'expected lifetime' ($L_{expected}$) and an 'estimated lifetime' ($L_{estimated}$). The 'expected lifetime' is a static calculation based on how many petabytes have already been written. For example, a brand new SSD is expected to last 5 years, but as it accumulates writes, the lifetime linearly shortens. The 'estimated lifetime' is a dynamic calculation of how long the SSD might last given the current application's write bandwidth. If the 'estimated lifetime' is smaller than the 'expected lifetime', then that is interpreted as a sign of high usage and accelerated endurance decay. Below are the two equations for this metric.

$$L_{expected} = (PBW_{rating} - PBW_{used}) \times \frac{5\ years}{PBW_{rating}} \tag{1}$$

$$L_{estimated} = \frac{PBW_{rating} - PBW_{used}}{B_{SSD}} \tag{2}$$

where PBW = petabytes written and B_{SSD} = write bandwidth to the SSD.

3.3.2 Performance Loss Estimation

Additionally, the CLC can be configured to monitor the dynamic performance loss of the application as a result of checkpointing to the SSD. If this option is enabled, the CLC monitors the amount of time elapsed since the launch of the program and the fraction of that time spent on checkpointing. We employ a stop-and-copy style checkpointing operation. Just before the next checkpoint, the CLC determines whether the time already lost to checkpointing exceeds the specified bound (e.g. 10%), and if so, directs the next checkpoint to the ramdisk. Each MPI process makes this decision independently.

$$T_{slowdown} = \frac{T_{chk}}{T_{compute} + T_{chk}} \tag{3}$$

3.3.3 Checkpoint Size

Finally, the CLC considers the size of the checkpoint to determine if there is enough ramdisk space available. Since ramdisk shares the main memory, its size must be limited to avoid swapping from the disk. CLC directs all large checkpoints to the SSD. However, if this decision conflicts with the prior 'lifetime' and 'performance loss' decisions, then the checkpoint is skipped altogether and the application moves on until the next checkpoint interval.

The downside to this approach is that it reduces the number of checkpoints and increases the average rollback distance during recovery. A more severe outcome is unintended uncoordinated checkpointing which can cause the application to restart from the beginning if all the MPI processes cannot agree on single synchronized checkpoint to roll back to. To avoid such issues, the CLC can potentially be forced to particular checkpoints.

3.3.4 *CLC Library*

Currently, the controller is written as a library that is added to the application's source code. It can interface with existing application-level checkpointing mechanisms and frameworks such as Scalable Checkpoint/Restart (SCR) [19]. The algorithm used by the controller is provided below. Lines 2-3 call the lifetime estimation and performance loss estimation features and lines 4-11 make a decision based on their results. Lines 12-15 checks the checkpoint size and skips writing large checkpoints to the ramdisk. Lines 16-17 actually writes the checkpoint and updates the checkpoint overhead measurement.

Algorithm 1 Checkpoint Location Controller (CLC)

```
1:  function CLC($D, r, i$) ▷ Where D - data, r - MPI rank, i - chkpnt number
2:      $L_{estimated}, L_{expected} = lifetimeEstimation()$
3:      $T_{slowdown} = performanceEstimation(T_{chk}, T_{total})$
4:      if $L_{estimated} > L_{expected}$ then
5:          $loc = SSD$
6:          if $T_{slowdown} > bound$ then
7:              $loc = RAM$
8:          end if
9:      else
10:         $loc = RAM$
11:     end if
12:     $size\_limit = TMPFS\_SIZE/numMPIRanks$
13:     if $loc == RAM$ and $sizeof(D) > size\_limit$ then
14:         return
15:     end if
16:     $writeCheckpoint(loc, D, r, i)$
17:     $update(T_{chk})$
18: end function
```

3.4 Recovery by Checkpoint Procedure

During restart, the application first searches for a checkpoint file that has been saved by a previous run. An attempt is always made to recover from the checkpoint in ramdisk. If it finds the latest checkpoint in ramdisk, it begins reading in that checkpoint. However, if the ECC logic signals a detectable, but uncorrectable memory error, then the entire ramdisk checkpoint is discarded. Information regarding uncorrectable memory errors can be located by 'edac' ('error detection and correction') kernel modules in Linux. The backup checkpoint file in the SSD is read in if the one in memory was corrupt. The checkpoint in the SSD could be older, leading to a longer rollback distance during recovery. We assume that the SSD has strong ECC built-in that protects its checkpoint and that it is always reliable.

4. ECC DESIGN

The proposed dual-ECC mode memory system has normal ECC for regular data, and strong ECC, that is Chipkill-Correct, for checkpoint data. A typical memory access to a DDR3 x4 memory module containing 18 chips (16 for data and 2 for ECC) reads out a data block of size 512 bits over 8 beats. A Chipkill-Correct scheme can correct errors due to a single chip failure and detect errors due to two chip failures. For x4 DRAM systems, such a scheme is based on a 4-bit symbol code with 32 symbols for data and 4 symbols for ECC parity and provides single symbol correction and double symbol detection. It has to activate two ranks with 18 chips per rank per memory access resulting in high power consumption and poor timing performance [20, 21]. In contrast, the proposed ECC schemes for regular and checkpoint data only activate a single x4 DRAM rank and have strong reliability due to the use of symbol-based codes that have been tailored for this application. Reed-Solomon (RS) codes are symbol based codes that provide strong correction and detection capability [22]. Here, we propose to use RS codes over Galois Field (2^8) for both normal and strong ECC modes.

Fault Model. When selecting the ECC algorithms for normal and strong ECC, the type of failures and how they manifest in the accessed data are considered. The DRAM error characteristics are well analyzed in [23, 24, 25]. In this work, we assumed errors are introduced by 5 different faults (bit/column/pin/word/chip) [26]. A bit fault leads to a single bit error in a data block. A column failure also leads to a single bit error in a data block. A pin failure results in 8 bit errors and these errors are all located in the same data pin positions. A word failure corrupts 4 consecutive bit errors in a single beat. A whole chip failure leads to 32 bit errors (8 beats with 4 bits/beat) in a 512 bit data block.

Faults can also be classified into small granularity faults (bit/column/pin/word) and large granularity faults (chip). Several studies have shown that small granularity faults occur more frequently than large granularity faults and account for more than 70% among all DRAM faults [23, 24, 25]. Hence, errors due to small granularity faults should be corrected with low latency in any ECC design.

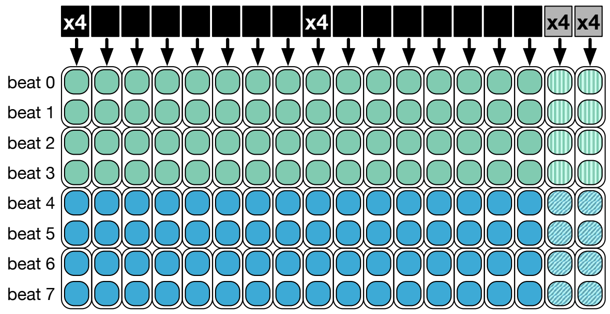


Figure 5: The depicted **normal ECC** access reads 512 bits from eighteen x4 chips, two of which are ECC chips. Two beats are paired up to create 1 8-bit symbol per chip. The first 4 and last 4 beats form two RS(36,32) codewords (green and blue).

4.1 Normal ECC

Normal ECC provides error correction coverage for regular data accesses, similar to typical ECC DIMMs for servers. It is designed to meet the following requirements:

1. To correct frequent errors due to single-bit/pin/word failures without triggering restart from a checkpoint.
2. To have small decoding latency of syndrome calculation since it is in the critical path of memory access.
3. To activate one rank per memory access and to have better timing/power/energy than Chipkill-Correct.

To satisfy these requirements, we use RS(36,32) over GF(2^8) for normal ECC. It has a storage overhead of 12.5%, which is the golden standard for ECC design [26]. RS(36,32) has a minimum distance of 5 and supports the following setups: (i) double error correction, (ii) four error detection, and (iii) single error correction and triple error detection [22]. If the decoder is designed for setup (i), then 2 symbol errors due to 1 chip failure can be corrected. However, 4 symbol errors due to 2 chip failures cannot be corrected and will lead to silent data corruption [26]. If designed for setup (ii), errors due to 2 chip failures can be detected but small errors due to bit/pin/word failures cannot be corrected. These small granularity faults are reported to occur frequently in memory systems and they must be corrected in order to avoid unnecessary restarts from checkpoints. Setup (iii) can correct all errors due to small granularity (single bit/pin/word) faults in a single chip, detect errors due to 1 chip failure, and has strong detection capability for 2 chip failures. Specifically, for double chip failures, setup (iii) can correctly detect several combinations of two small granularity faults and provide very strong detection for the other cases. Based on this reasoning, RS(36,32) with setup (iii) is chosen to protect normal data.

Results will later show that the normal ECC scheme has a very low silent data corruption rate of 0.003% and a small latency of 0.48ns for the syndrome calculation. Furthermore, since only 1 rank is activated in each memory access, it has better timing/power/energy performance than the traditional x4 Chipkill-Correct scheme.

Memory access pattern: As illustrated in Figure 5, upon a memory read, one rank with 18 chips are activated and 512 bits are read out over 8 beats. Each beat contains 4 bits from a single chip, thus two beats can be paired to form an 8-bit symbol in an RS codeword. The $18 \times 2 = 36$ symbols from the first 4 beats are sent to one RS(36,32) decoding unit followed by the second set of 36 symbols from the next 4 beats. If a codeword has 1 symbol error, it is corrected and sent to the last level cache (LLC). If an uncorrectable error (i.e. >1 erroneous symbol) is encountered, then a flag is set. In such a case, the OS would see the flag, terminate the application, and trigger rollback and restart from the checkpoint. Upon a memory write, the ECC encoder forms two RS(36,32) codewords and stores them in a DRAM rank as in a normal memory write.

4.2 Strong ECC

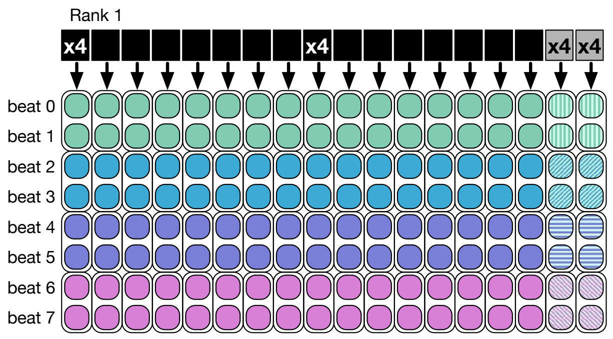
Checkpoints that are stored in DRAM memory have to be protected by a strong ECC mechanism to preserve the integrity of the checkpoint data. The proposed strong ECC is designed to meet two requirements:

1. To provide Chipkill-Correct level reliability, which can correct all errors due to a single chip failure and detect all errors due to two chip failures. The strong error correction capability reduces the probability of accessing the SSD's checkpoint during restart.
2. To require minimal differences in hardware so as to be able to switch easily from and to normal ECC. Since ramdisk pages can be mapped anywhere in physical memory, the DRAM modules should be flexible in holding normal or checkpoint data without special modifications to the DRAM devices.

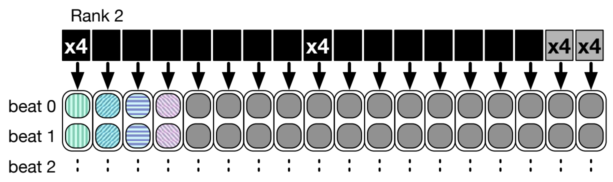
We propose using RS(19,16) over GF(2^8) for strong ECC. It works by a hierarchical two-layer scheme where 18 out of the 19 symbols are stored in one rank and the 19th symbol (the third parity symbol) is stored in another rank, as in V-ECC [27].

The two-layer scheme works because of the embedded structure of the RS code [22]. The parity check matrix of RS(18,16) is embedded in the parity check matrix of RS(19,16) and thus these two codes can share the same decoding circuitry. The two symbols in the syndrome vector of RS(18,16) are identical to the first two symbols in the syndrome vector of RS(19,16). Once RS(18,16) detects errors, the third ECC symbol can be used to generate the third symbol of the syndrome vector of RS(19,16) and then the RS(19,16) decoder can perform error correction [22].

A direct implementation of this scheme would result in two memory accesses thereby degrading performance and incurring higher power consumption. Thus, an ECC cache is employed to store the third parity symbol and hide the latency due to the extra read and write accesses as in [27]. Additionally, activating just one rank per memory access has better timing/power/energy compared to conventional Chipkill-Correct.



(a) Strong ECC, Memory access 1



(b) Strong ECC, Memory access 2

Figure 6: (a) Strong ECC creates four RS(18,16) codewords (green, blue, purple, and pink); each codeword is based on 2 beats of data; (b) If errors are detected, four additional ECC symbols are retrieved to form four RS(19,16) codewords.

Memory access pattern: As illustrated in Figure 6a, upon a memory read, only one rank is activated and 18 symbols (16 data + 2 ECC) are sent to the RS(18,16) decoder. Every two beats of data form one RS(18,16) codeword. The RS(18,16) decoder is designed to perform detection only. Note that RS(18,16) can detect up to 2 symbol errors (2 chip failures). If it detects errors, the decoder is halted and the third parity symbol is fetched from the ECC cache and sent to the RS(19,16) decoder. If the ECC cache does not have the parity symbol, then a second memory access is used to get it from another rank (Figure 6b). RS(19,16) can perform single symbol correction and double symbol detection (SSC-DSD) and can thus provide Chipkill-Correct level protection. If the RS(19,16) decoder detects an uncorrectable error, then the entire DRAM checkpoint is discarded and the application retrieves a potentially older checkpoint from the SSD. The recovery procedure was outlined in Section 3.4.

Upon a memory write, 512 data bits are encoded into 4 codewords. Two of the parity symbols in each codeword are stored in the two ECC chips in the same rank by a regular memory write operation. The third parity symbol is stored in the ECC cache or in another DRAM rank.

4.3 Modification to the Memory Controller

The strong ECC mode exists simultaneously with normal ECC that protects regular memory data; and only requires modification to the memory controller, not the DRAM devices. In order to identify ramdisk/checkpoint data, the page table can be marked with a special flag to indicate ramdisk pages. As illustrated in Figure 7, regular data is routed via the normal encoder/decoder and ramdisk data is routed via the strong encoder/decoder. We rely on an ECC address translation unit to determine the location of the second memory access for strong ECC as in V-ECC [28].

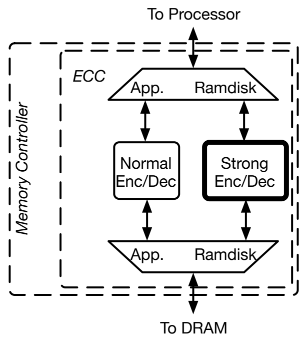


Figure 7: Modified Memory Controller with two decoders for normal and strong ECC.

5. EVALUATION SETUP

5.1 Microbenchmark

A microbenchmark was written to evaluate the performance of writing a wide variety of checkpoint sizes to different platforms. It was written as an MPI program in C++ to simulate typical parallel supercomputing applications. It mainly consists of two phases: compute and checkpoint. The compute phase runs an algorithm which takes roughly 5 seconds to finish, and the checkpoint phase writes a file of a specified size to either the ramdisk or the SSD. The microbenchmark consists of 100 total iterations of the two phases.

The microbenchmark can be launched with any desired number of MPI processes. To take our measurements, we ran the microbenchmark with 64 MPI processes across 8 nodes. The desired checkpoint size is passed into the microbenchmark as an input, and the same size of checkpoint is made in all 100 iterations. Although there are some supercomputing applications whose checkpoint sizes vary during runtime, most applications save a particular data structure such as the <x,y,z> position vectors of particles or a vector of temperatures. Thus, having a fixed checkpoint size throughout is acceptable.

We measured the total runtime of the microbenchmark under three naïve implementations i) no checkpointing, ii) checkpointing to ramdisk only, and iii) checkpointing to SSD only. The results, which were already shown in Figure 1 in Section 2, indicated that writing the checkpoint to ramdisk incurs only a small slowdown of 14%, whereas the SSD incurs a 4.6× slowdown.

5.1.1 *Typical Checkpoint Sizes*

Checkpoint sizes can be reported for an MPI process, for a node, or for an entire application. It is difficult to determine real checkpoint sizes unless real HPC applications are run at scale on a supercomputer. Even though mini-apps and proxy-apps are representative of the algorithms of the HPC applications, one of their shortcomings is that they are not representative of the runtime or the memory size of large HPC applications.

We conducted a survey of past literature to determine typical checkpoint sizes. An older version of NAS Parallel Benchmark suite checkpointed 3.2MB-54MB per process [29]. MCRENGINE, a checkpoint data aggregation engine, was evaluated on applications having checkpoint sizes between 0.2MB-154MB per process [30]. An experiment on Sierra and Zin clusters at LLNL wrote 50MB and 128MB per process, respectively [31]. A PFS-level checkpointing evaluation on two large clusters HERMIT and LiMa wrote 294MB and 340MB per process, respectively [32]. Note that often times more than one MPI process runs on a multi-core node. Node level checkpoint sizes have been reported between 460MB-4GB/node [16].

To illustrate the wide variety of existing checkpoint sizes, our microbenchmark experiments use between 100MB-1000MB per MPI process; and we run 8 MPI processes per node.

5.2 Proxy-apps

The proposed Checkpoint Location Controller was validated against two real benchmarks: *miniFE* and *Lulesh*. *miniFE* is a proxy-app whose main computation is solving a sparse linear system using a conjugate-gradient (CG) algorithm. In a checkpoint, miniFE saves solution and residual vectors. *Lulesh* is a proxy-app that models shock hydrodynamics. It solves a Sedov blast problem while iterating over time steps. In a checkpoint, Lulesh saves the vectors for energy, pressure, viscosity, volume, speed, nodal coordinates, and nodal velocities. The simulation setup and the parameters used to run the benchmarks are given in Table 2. The parameters were decided upon using instructions that came with each application on how to scale up the problem size given the available memory in each node, which was 24GB in our servers.

Table 2: Simulations parameters for *miniFE* and *Lulesh*

	miniFE	Lulesh
Parameters	528×512×768	45×45×45
Setup	64 MPI processes, 8 nodes 24GB/node	
Checkpoint sizes:		
1 MPI proc:	50 MB	8 MB
1 node:	400 MB	64 MB
App. Total:	3.1 GB	512 MB
Baseline runtime:	236 sec.	74,470 sec.
Checkpoint behavior:	once/iteration, ~1 sec/iter, 200 iterations,	once/iteration, ~11 sec/iter, 6,499 iterations

5.3 SSD Device Reference

We chose the Intel DC S3700 SSD in 800 GB using a SATA 3 6Gbps connection for our experiments [4]. It reported an endurance rating of 14.6 PBW and a maximum sequential write speed of 460 MB/s. We were able to achieve write speeds of only 250 MB/s during our checkpoint experiments. The write bandwidth to the SSD is important because faster writes lead to less application slowdown and less overall power consumption. There is a PCIe version of the same SSD available with higher bandwidth; however PCIe is more expensive. On CDW-G, a popular IT products website, Intel's PCIe-based SSDs for data centers retail at upwards of 92¢ per gigabyte. On the other hand, their SATA SSDs retail as low as 71¢ per gigabyte.

6. RESULTS

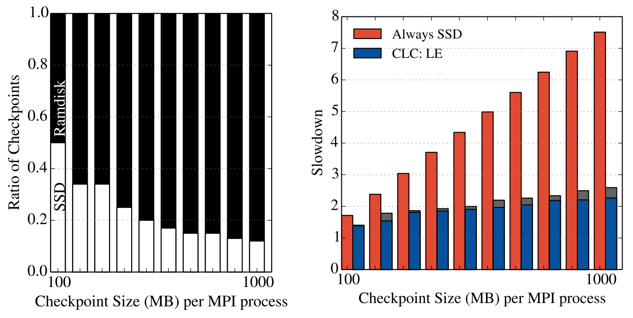
6.1 Controller Results

6.1.1 *Lifetime Estimation Results*

The first set of results are with only the lifetime estimation (abbreviated **LE**) feature. Again, the controller uses Eq. 1 and Eq. 2 (Section 3.3.1) before each checkpoint to determine if the current rate of checkpointing by the application will prematurely wear out the SSD. We assumed an endurance rating of 14.6 PBW (on a brand new SSD) that leads to 5 years of useful life.

Each node has a local SSD and the controller takes into account the endurance of the local SSD and the cumulative bandwidth of 8 MPI processes in the node writing checkpoint files to it. As can be seen in Figure 8a, once the endurance is taken into account, fewer checkpoints are written to the SSD, especially at larger checkpoint sizes. At 1000MB per process, only 12% of checkpoints are written to the SSD. Advantageously, this leads to a performance improvement; the slowdown of the benchmark is considerably lessened to an average of only 1.9× (Figure 8b)—as opposed to the nearly 8× slowdown (4.6× on average) if always checkpointing to the SSD.

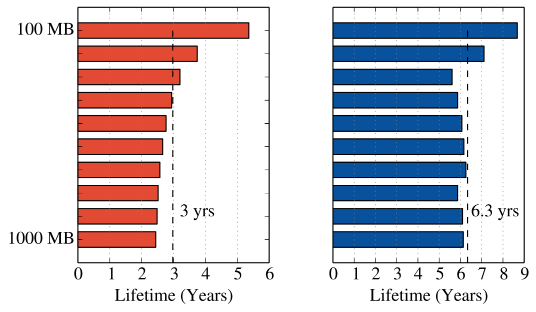
The shaded region above each bar for the CLC's results in Figure 8b indicates the overhead due to encoding the checkpoint data with strong ECC before writing to the DRAM. In experiments, the overhead of a second memory access was simulated by writing the checkpoint twice to DRAM. Using this method to measure ECC overhead predicted about 20% additional slowdown, making the average slowdown about 2.1×. This is a worst case estimation of the ECC overhead; in practice, the second memory access can be optimized by using an ECC cache for parity symbols of strong ECC.



(a) Checkpoint Location (b) Slowdown

Figure 8: Microbenchmark results with the CLC's lifetime estimation (LE) feature enabled. (a) For bigger checkpoint sizes, more checkpoints are written to the ramdisk. (b) The CLC significantly reduced the slowdown. The shaded region above each bar is the overhead for strong ECC's second memory access.

Figure 9 shows the improvement in endurance gained by the endurance-aware checkpoint controller. This result was obtained after the application completed, and was based on its runtime and how many checkpoints it wrote to the SSD. If checkpoints were only written to the SSD as in Figure 9a, then the SSD is estimated to last an average of 3 years across all the checkpoint sizes. On the other hand, the LE feature of the controller extended the SSD lifetime to an average of 6.3 years (Figure 9b), ensuring that users can get the guaranteed 5 years of life from their SSD.



(a) SSD Only (b) Controller: LE

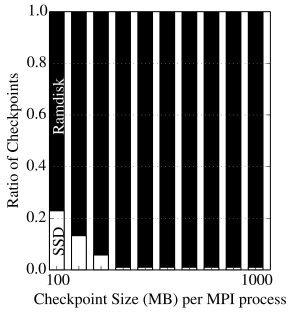
Figure 9: Expected lifetime of the SSD is improved with the LE feature in the CLC.

6.1.2 *Performance Estimation Results*

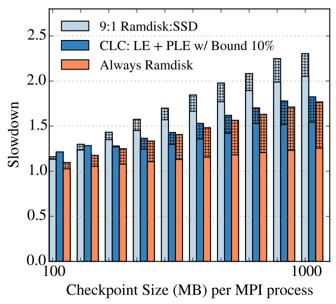
Although, the controller was able to successfully prolong SSD endurance, the application still experienced 2.1× slowdown, as was shown in Figure 8b. To further minimize performance loss, with the LE feature still enabled, we also enabled the performance loss estimation (abbreviated **PLE**) feature. The performance loss bound was set to 10% in this experiment. Note that the 10% bound was optimistic because even the 'always-ramdisk' checkpoint experienced 3%-25% slowdown across the checkpoint sizes.

As Figure 10a shows, the controller wrote even fewer checkpoints to SSD when the PLE feature was enabled; almost 99% of checkpoints were written to ramdisk. Nevertheless, it was successful in decreasing slowdown even further to only 36% on average (47% with strong ECC overhead). More importantly, the controller's achieved performance is more closer to the 'always-ramdisk' approach which achieved 14% slowdown on average (42% with strong ECC overhead).

For the sake of comparison, we also implemented a naïve scheme where every 10th checkpoint is written to the SSD, labeled as "9:1 Ramdisk:SSD" in Figure 10b. This scheme performed better than CLC's smarter scheme for checkpoint sizes of 100 MB and 200 MB per process, indicating that a fixed scheme might be sufficient for applications with small checkpoint sizes that want to achieve a balance between performance and reliability. However, across all checkpointed sizes, it's average slowdown was 58% (72% with strong ECC overhead), that is 22% worse than CLC's PLE feature. The ratio 9:1 was arbitrarily picked; a larger ratio can be chosen for even smaller performance loss if the DRAM checkpoint has strong ECC protection.



(a) Checkpoint Location



(b) Slowdown

Figure 10: (a) Performance loss estimation (PLE) feature attempts to contain the performance loss within a specified bound (e.g. 10%) and leads to even fewer checkpoints to the SSD. (b) PLE's improved slowdown is closer to ramdisk's. Shaded regions above each bar represent worst-case overheads from strong ECC encoding.

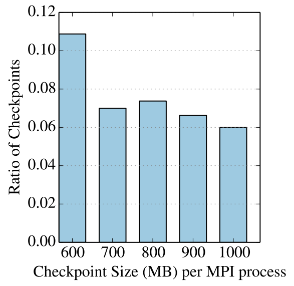
6.1.3 *Size Results*

CLC's size checking feature is configured to direct checkpoints bigger than a particular size (e.g 0.5GB) to the SSD, that is, these large checkpoints are never written to the ramdisk. In this configuration, some checkpoints maybe skipped if the LE and PLE features indicate unfavorable results. Figure 11a shows that for checkpoint sizes 600MB and bigger, the CLC wrote less than 10% of the intended number of checkpoints. It also increased the average checkpoint interval of this microbenchmark (ideally a 5-second interval) from less than 10 seconds to 1-2.5 minutes (Figure 11b).

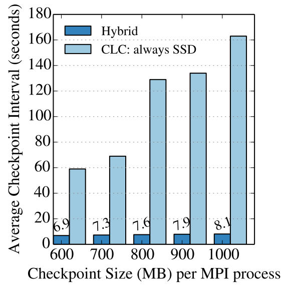
Skipping checkpoints leads to longer rollback distances and, more severely, to unintended uncoordinated checkpointing (Section 3.3.3). To avoid such issues, the CLC can be changed forcefully write particular checkpoints.

6.1.4 *Energy Results*

Energy saved from writing checkpoints to the DRAM is an additional benefit of our proposed hybrid method. First, we measured the power consumed during a checkpoint operation to both the ramdisk and the SSD. Power measurements were obtained via the "watts up?" meter and its smallest



(a) Checkpoints Made to SSD



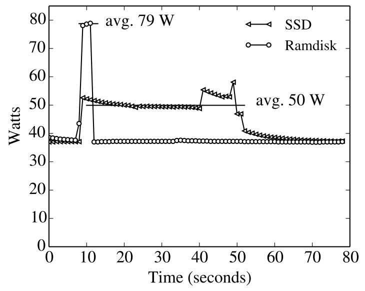
(b) Avg. Checkpoint Interval

Figure 11: (a) With CLC's size checking feature, big checkpoints are always written to the SSD. But this leads to only a small fraction of checkpoints actually being written, while the rest are skipped. (b) This feature drastically increases the average checkpoint interval.

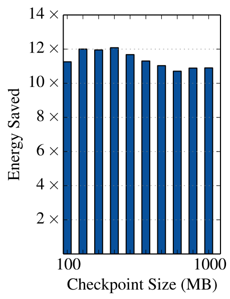
sampling rate is 1 second. It measures the load of one entire server node; thus, the measured power includes everything from CPU, DRAM, I/O bus, SSD, and more.

Figure 12a shows the node's power consumption while continuously writing a 10GB file. We chose a very large file size to obtain a measurable power sample because writing small files to the DRAM is very fast (under 1 second) and does not get picked up by the "watts up?" meter. The idle power of the server was 37W and checkpointing to the SSD saw a jump to 50W on average. Interestingly, checkpointing to DRAM registers much higher power consumption at 79W on average. However, writing to the DRAM took only 3 seconds compared to the 42 seconds for the SSD. Overall, DRAM uses less energy because of its speed advantage.

Second, the power numbers obtained from the power profile and the ratio of checkpoints sent to the ramdisk vs. SSD were used to calculate the total energy consumption during checkpointing. Figure 12b shows that between 10×-12× energy savings were gained from the checkpoints that were written to the ramdisk instead of the SSD. These results demonstrate the energy savings with only the LE feature from Figure 8.



(a) Power Profile



(b) Energy Savings

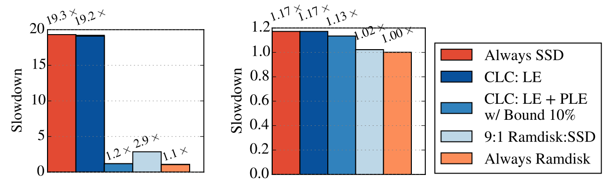
Figure 12: (a) The SSD consumes 50W during a write operation, whereas the DRAM consumes 79W. (b) However, due to DRAM's faster write bandwidth, re-directing some checkpoints to the DRAM saves overall checkpoint energy.

6.1.5 *Real Application Results*

Finally, we evaluated the CLC with real benchmarks *miniFE* and *Lulesh*. *miniFE* wrote 50MB checkpoints per

MPI process with only 1 second of computation in a checkpoint interval. At a node level, 8 MPI processes write 400MB of checkpoints each iteration. As can be seen in Figure 13a, the 'always-SSD' approach caused nearly a 19× slowdown, as did the CLC with LE feature enabled. The slowdown is a consequence of the frequent checkpoint behavior of this application. However the checkpoints were small enough not to cause premature wearing out of the SSD; hence, the CLC directed almost all checkpoints to the SSD. Enabling the PLE feature with a bound of 10% was able to decrease the slowdown to 1.2×, but then the CLC directed almost all checkpoints to the ramdisk. In comparison the "9:1" scheme that sent 1 out of 10 checkpoints to the SSD saw a 2.9× slowdown and 'always-ramdisk' approach saw a 1.1× slowdown.

Figure 13b shows the results for *Lulesh*, which wrote very small 8MB checkpoints per MPI process at a sufficiently large interval of 11 seconds. Since the bandwidth to the SSD is low enough so as not to cause accelerated endurance decay, the CLC always chose the SSD. Enabling the PLE feature reduced the performance loss from 17% down to 13% by redirecting 17% of all checkpoints to the ramdisk. With only 2% slowdown, Lulesh is an example of an application that might be better suited for a static "9:1" scheme that balances out both reliability and performance.



(a) miniFE (b) Lulesh

Figure 13: Neither miniFE nor Lulesh checkpoints with high enough bandwidth to wear down the SSD; thus CLC's LE feature allows most checkpoints to the SSD. Enabling the PLE feature, on the other hand, makes the CLC re-direct most of miniFE's checkpoints to the DRAM.

6.2 ECC Overhead & Error Coverage

The performance overhead of ECC on application runtime were already included in results in Figures 8-13. This section focuses on synthesis and error coverage results for ECC.

6.2.1 *Synthesis Results*

We synthesized the decoding units of RS(36,32), RS(18,16) and RS(19,16) codes over GF(2^8) using 28nm industry library. The syndrome calculation is performed for every read and so we optimize it for very low latency. The decoding latency of syndrome calculation is 0.48ns for RS(36,32) code and 0.41ns for RS(18,16) and RS(19,16) codes. Thus the syndrome calculation latency is less than one memory cycle (1.25ns if the DRAM frequency is 800MHz).

For normal ECC, if syndrome vector is not a zero vector, RS(36,32) performs single symbol correction and triple symbol detection. It takes an additional 0.47ns to correct a single symbol error or declare that there are more errors. For strong ECC, RS(18,16) is configured to only perform detection. If the syndrome vector is not a zero vector, the memory controller reads the third ECC symbol and forms the RS(19,16) code. After calculating the syndrome vector for RS(19,16), the decoder spends an additional 0.47ns to correct a single symbol error and if it cannot correct the error, it declares that there are more errors. The synthesis results are shown in Table 3.

Table 3: Synthesis results for proposed RS codes

	RS(36,32)	RS(18,16)	RS(19,16)
Syndrome Calculation	0.48ns (σ)	0.41ns (ρ)	0.41ns (ρ)
Single Symbol Correction & Double Symbol Detection	N/A	N/A	ρ + 0.47ns
Single Symbol Correction & Triple Symbol Detection	σ + 0.47ns	N/A	N/A

Table 4: The error protection capability

Failure Mode	RS(36,32)	RS(18,16)	RS(19,16)	Chipkill-Correct
Single Chip Failures				
1 bit	DCE: 100% DUE: 0% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%
1 pin	DCE: 100% DUE: 0% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%
1 word	DCE: 100% DUE: 0% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%
1 chip	DCE: 100% DUE: 0% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%	DCE: 100% DUE: 0% SDC: 0%
Double Chip Failures				
1 bit + 1 bit	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 bit + 1 pin	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 bit + 1 word	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 bit + 1 chip	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 pin + 1 word	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 pin + 1 pin	DCE: 0% DUE: 99.9999% SDC: 0.0001%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 pin + 1 chip	DCE: 0% DUE: 99.9969% SDC: 0.0031%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 word + 1 word	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 word + 1 chip	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%
1 chip + 1 chip	DCE: 0% DUE: 99.9969% SDC: 0.0031%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%	DCE: 0% DUE: 100% SDC: 0%

6.2.2 *Error Coverage*

The reliability of four ECC schemes, namely, RS(36,32) for normal ECC, RS(18,16) and RS(19,16) for strong ECC, and x4 Chipkill-Correct was evaluated. 10 million Monte Carlo simulations for single bit, pin, word, and chip failure events were conducted. Each fault type was injected into a single chip or two chips. For each type of error event, the corresponding detectable and correctable error (DCE) rate, detectable but uncorrectable error (DUE) rate and silent data corruption (SDC) rate [26] were calculated; Table 4 gives the corresponding simulation results for these four ECC codes.

RS(36,32) for normal ECC can correct all errors due to small granularity faults and can detect all errors due to a single chip failure. For faults across 2 chips, it can fully detect errors due to a single bit fault in each chip, a single bit fault in one chip and a single pin fault in another chip, and several other error events as shown in Table 4. This code has good detection capability for errors due to a pin fault in each chip, 1 pin fault in one chip and 1 chip failure and double chip failures.

The combination of RS(18,16) and RS(19,16) that is used for strong ECC achieves Chipkill-Correct reliability. Recall that RS(18,16) is activated every time to provide detection. It can detect all errors due to double chip failures, and once errors are detected, RS(19,16) decoder is activated. It can correct all errors due to a single chip failure and detect errors due to double chip failures and thus it achieves Chipkill-Correct level reliability.

7. RELATED WORK

Zheng et. al [14] proposed to pair two processors in a *buddy* system where each process makes two identical checkpoints to its own local storage and to the buddy's local storage. The default local storage is the local memory, known as *double in-memory checkpointing*; if a local disk is available then *double in-disk checkpointing* can be carried out instead. At recovery, one of the two buddies provides the restoration checkpoint. Similar to our results, their in-memory checkpoint was faster, but the disk was more practical for applications with big memory footprints. We believe that our two methods can be combined to form a better hybrid-buddy checkpointing method where instead of wasting memory by storing double checkpoints to attain resilience, either our ramdisk or SSD checkpoint can be stored at the buddy's node.

Rajachandrasekar et. al [33] proposed a new in-memory file system called CRUISE (**C**heckpoint **R**estart in **U**ser **S**pac**E**) in which large checkpoints to main memory can transparently spill over to SSD storage. CRUISE is mounted similarly to a ramdisk. Our work can augment CRUISE by providing the necessary strong ECC protection for memory checkpoints. Similarly, CRUISE's spill feature can augment our CLC for checkpoints that are too large to fit in memory. CLC's lifetime estimation feature can provide CRUISE with important information about the endurance of the SSD/spill device.

Saito et al. [15] investigated improving energy consumption during checkpoint write operations to a PCIe-attached NAND-flash device. They suggest that there exists an optimal number of I/O processes that can simultaneously write to the device. They minimize energy consumption by applying DVFS and keeping an I/O profile that helps to quickly determine the optimal number of I/O processes. This work could possibly be added to our CLC as a new "energy estimation" feature and help predict energy consumption for an energy-limited system that checkpoints to SSDs.

Yoon and Erez [28] proposed Virtual ECC (V-ECC) to protect memory systems with strong ECC mechanisms without modifying existing DRAM packages. This idea makes it possible to provide large parity even for systems that have no dedicated parity hardware. We borrow their technique to provide strong ECC protection for our checkpoints, where the extra parity symbols for strong ECC is stored like data.

8. CONCLUSION

Exascale supercomputers have millions of components that can fail. A 100 petabyte memory system—100× larger than ORNL Titan supercomputer's 1 petabyte memory system—alone consists of millions of DDR4 DRAM devices backed by hundreds of thousands of SSD flash devices. Resilience to failing components must be achieved by creating a fast and reliable checkpoint/restart framework.

In this paper, we proposed a hybrid DRAM-SSD checkpointing solution to achieve speed and reliability for local checkpointing while also reducing the endurance decay of SSDs. The Checkpoint Location Controller (CLC) that we implemented monitors SSD endurance, performance degradation, and checkpoint size to dynamically determine the best checkpoint location. CLC running on a microbenchmark showed an SSD lifetime improvement from 3 years to 6.3 years. Application results on *miniFE* and *Lulesh* validated that the online controller can make appropriate decisions to limit the slowdown due to checkpointing.

Furthermore, our normal ECC provides low-latency correction for errors due to bit/pin/column/word faults and our strong ECC provides Chipkill-Correct capability to DRAM checkpoints to reduce the overheads of rollback. The system presented in this paper demonstrates that it is in fact possible to build an exascale memory system using commodity DRAM and SSD and gain both speed and reliability without relying on emerging memory technologies.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their time and input. We also thank the generous support of our industrial sponsor, ARM Ltd.

References

[1] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, 2009.

[2] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, 2007.

[3] Xiangyong Ouyang, S. Marcarelli, and D.K. Panda. Enhancing checkpoint performance with staging io and ssd. SNAPI 2010.

[4] Intel Cooperation. Intel Solid-State Drive DC S3700 specification, October 2012.

[5] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems. SC 2009.

[6] Ping Chi, Cong Xu, Tao Zhang, Xiangyu Dong, and Yuan Xie. Using Multi-level Cell STT-RAM for Fast and Energy-efficient Local Checkpointing. ICCAD 2014.

[7] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic. Optimizing Checkpoints Using NVM as Virtual Memory. IPDPS 2013.

[8] U.S Department of Energy Office of Science and National Nuclear Security Administration. Preliminary Conceptual Design for an Exascale Computing Initiative, November 2014.

[9] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. MSST 2012.