Real-Time In-Memory Checkpointing for Future Hybrid Memory Systems

Shen Gao[1,2] Bingsheng He[1] Jianliang Xu[2]
[1]Nanyang Technological University, Singapore
[2]Hong Kong Baptist University

ABSTRACT

In this paper, we study real-time in-memory checkpointing as an effective means to improve the reliability of future large-scale parallel processing systems. Under this context, the checkpoint overhead can become a significant performance bottleneck. Novel memory system designs with upcoming non-volatile random access memory (NVRAM) technologies are emerging to address this performance issue. However, we find that those designs can still have prohibitively high checkpoint overhead and system downtime, especially when checkpoints are taken frequently to implement a reliable system. In this paper, we propose a novel in-memory checkpointing system, named *Mona*, for reducing the checkpoint overhead of hybrid memory systems with NVRAM and DRAM. To minimize the in-memory checkpoint overhead, Mona dynamically writes partial checkpoints from DRAM to NVRAM during application execution. To reduce the interference of partial checkpointing, Mona utilizes runtime idle periods and leverages a cost model to guide partial checkpointing decisions for individual DRAM ranks. We further develop load-balancing mechanisms to balance checkpoint overheads across different DRAM ranks. Simulation results demonstrate the efficiency and effectiveness of Mona in reducing the checkpoint overhead, downtime and restarting time.

Categories and Subject Descriptors

D.4.5 [**Reliability**]: Checkpoint/restart; C.4.1 [**Computer Systems Organization**]: Performance of Systems–design studies

General Terms

Design, Performance

Keywords

NVRAM, Phase Change Memory, Checkpointing, Parallel Computing

1. INTRODUCTION

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). Copyright is held by the author/owner(s).
ICS'15, June 8–11, 2015, Newport Beach, CA, USA.
ACM Copyright 2015 ACM 978-1-4503-3559-1/15/06...$15.00
http://dx.doi.org/10.1145/2751205.2751212 .

The increasing scale of parallel processing systems is being severely challenged by high failure rates. It is predicted that transient errors in CPUs and memories will increase by 32 times in the next decade [3]. On the other hand, increasingly large memory capacities cause a higher chance of memory failures [28]. As a result, the upcoming large-scale system will have the severe problem of frequent crashes, which can degrade the system performance dramatically. Real-time in-memory checkpointing offers a fine-grained checkpointing, which can be an effective means to improve the reliability of future large-scale parallel processing systems. In the event a failure occurs, the application is restarted from the latest checkpoint, thus minimizing repeated computation. But, due to potential high failure rates in large-scale parallel processing systems, checkpoints should be taken in a real-time and fine-grained manner. Obviously, high checkpointing frequency poses significant challenges on novel checkpoint mechanism designs. Thus, this paper investigates whether and how we can reduce the overhead of checkpointing-restart mechanisms to enable real-time in-memory checkpointing in large-scale systems.

Traditionally, hard disk-based checkpoint systems have been adopted. Since hard disks are over two orders (even three orders) of magnitude slower than main memory, checkpoint/restart mechanisms can cause significant overhead to the overall system performance. A large-scale system may suffer more than 50% performance degradation and spend up to 80% of extra I/O traffic, because of frequent checkpointing operations [22, 11, 10]. Even worse, during checkpointing, the application execution may suffer from a period of downtime for writing checkpoints. This downtime is exaggerated for applications with large memory footprint.

More recently, emerging NVRAM technologies such as phase-change memory (PCM) have been exploited to reduce the checkpoint overhead (i.e., NVRAM-assisted checkpoint) [6, 7, 33]. Those studies have demonstrated promising results in reducing the checkpoint overhead of large-scale systems. However, existing NVRAM-assisted checkpoint techniques [6, 7, 33] have the following three limitations towards real-time in-memory checkpointing. First, even with NVRAM, they still cause a long downtime. This is partly because the low latency feature of PCM is not exploited to reduce the total checkpoint overhead (details are presented in Section 3.1). Second, while asynchronous or non-blocking checkpointing can reduce the downtime to a certain degree, the mechanism is complicated and the runtime overhead can be high in practice [27]. Third, most previous studies (such as [6, 7, 33, 14, 16]) target at

a sole optimization goal. Some applications may simply require minimizing the checkpoint overhead so that the total expected execution time is minimized. Other critical applications might have rigid requirements on the restarting time or downtime. System re-design/re-implementation is required for a different optimization goal.

In this paper, we propose a novel hybrid memory design, named *Mona*, towards enabling real-time in-memory checkpointing. Particularly, Mona uses PCM, one of the promising NVRAM technologies, to store checkpointing files. Mona has the following novel designs to address the aforesaid limitations of the existing techniques:

- To support efficient checkpointing-restart mechanisms, we develop a PCM-assisted dynamic checkpointing approach to perform partial checkpointing from DRAM to PCM. Additionally, our design takes advantage of memory rank parallelism. Inspired by speculative checkpointing [18], partial checkpointing is performed at application runtime.
- To minimize the interference, we write dirty pages from DRAM to PCM *only when* there are sufficiently lengthy idle periods in a DRAM rank. Instead of determining the length of each idle period, we estimate the idle period distributions and determine a *write threshold* on the idle period length: only when the current idle period exceeds the write threshold, we trigger the dirty-page writing from DRAM to PCM. That can avoid the penalty caused by checkpointing in very short idle periods. Also, we carefully select the dirty pages to be written to PCM.
- Balancing the checkpoint overhead across different memory ranks is another important performance issue for Mona, since writes to PCM are often a performance bottleneck in hybrid memory systems [25]. We propose load-balancing strategies to minimize the worst checkpoint overhead among all DRAM ranks. Both PCM-assisted dynamic checkpointing and load-balancing strategies are guided by our cost model for adapting idle period distributions and page access patterns. With the cost model, Mona can optimize the system for different goals such as the total checkpoint overhead or the restarting time.

We implement our design into a cycle-accurate simulator and evaluate the efficiency and effectiveness of Mona with both micro and macro benchmarks. Overall, Mona significantly reduces the checkpoint overhead, downtime and restarting time. Particularly, compared with the state-of-the-art PCM-based checkpointing method [6], Mona reduces the total checkpointing cost by 65%, and the restarting time by 80%. We extend Mona to support multi-level checkpointing and demonstrate the effectiveness of our approach in multi-level checkpointing for larger-scale applications.

Organization. The rest of the paper is organized as follows. We briefly introduce the background on NVRAM and checkpointing in Section 2. Section 3 describes an overview of Mona, followed by detailed design and implementations in Section 4. The experimental results are presented in Section 5. We review the related work in Section 6 and conclude this paper in Section 7.

2. PRELIMINARY AND BACKGROUND ON NVRAM

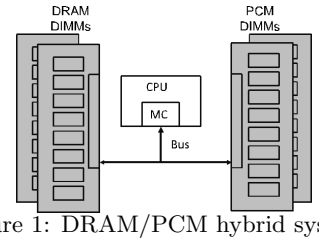


Figure 1: DRAM/PCM hybrid system.

Since conventional DRAM technologies encounter scaling difficulties and power issues, NVRAM technologies such as PCM and memristors have been developed. This paper focuses on PCM, which has gained much attention in the research community [6, 34, 15, 12]. The read latency of PCM is close to that of DRAM and two orders of magnitude shorter than that of flash memory. The write latency of PCM is in between those of DRAM and flash memory. Without erase-before-write constraints, its random-write performance is much better than that of flash memory. Moreover, PCM is likely to be cheaper than DRAM when produced in mass market quantities. It is feasible that PCM will be integrated into future computer systems.

Recent studies have proposed three major methods of integrating PCM into the memory hierarchy. The first approach is to use PCM as a replacement of DRAM [15, 2]. The second approach is to use PCM as an external storage like a hard disk [5]. The third approach is to integrate PCM into DIMMs and the main memory system becomes a hybrid system [25, 6, 8]. This paper adopts the third approach. PCM is allocated on separate DIMMs for a better bandwidth and ease-of-integration, as illustrated in Figure 1. In this study, we assume a DDR3 memory system, where PCM and DRAM are allocated on DDR3 DIMMs. Each DIMM contains multiple ranks to transmit data independently.

Our design is inspired by NVDIMM [20], which is commercially available and is a combination of DRAM and NAND flash. During normal operations, NVDIMM is working as DRAM while flash is invisible to the host. However, upon power failure, NVDIMM saves all the data from DRAM to flash by using supercapacitor to make the data persistent. In contrast, our design leverages PCM to replace NAND flash in NVIDIMM, eliminating supercapacitors. Moreover, since PCM is much faster than NAND flash, our design enables real-time in-memory checkpointing.

3. SYSTEM DESIGN

In this section, we discuss design motivations and give a design overview of Mona.

3.1 Design Motivations

Existing NVRAM-assisted checkpoint techniques [6, 7, 33] all use synchronous and incremental checkpointing. At each checkpointing stage, the application execution is temporarily suspended, and all the dirty pages in DRAM are written to PCM. The role of PCM is similar to that of hard disks in the sense that both of them are used as persistent storage for storing checkpointing files, and little modifications are required for the conventional checkpointing-restart algorithm. However, such a simple approach has several obstacles/deficiencies:

- First, during the checkpointing process, it has to write many dirty pages from DRAM to PCM, still causing a significant overhead and downtime especially on large-memory machines. Another issue is on memory ranks.

To fully utilize the memory bandwidth, all memory ranks should perform checkpointing in parallel. The checkpoint overhead is thus bounded by the longest writing latency of all the ranks. Uneven loads among the ranks would magnify the checkpointing cost.

- Second, it may underutilize the PCM bandwidth. On one hand, PCM is only used in checkpointing, and is idle during application execution. On the other hand, memory-intensive workloads still have many idle periods with reasonable lengths (e.g., hundreds of cycles) [13, 31, 17]. It is desirable to take advantage of those idle periods.
- Finally, as architectures and applications vary, we should provide the flexibility to achieve different optimization goals, such as minimizing either the total checkpointing cost or the restarting time.

3.2 Overview of Mona

To address the aforementioned obstacles, we propose a novel in-memory checkpointing design Mona with *dynamic partial checkpointing* and cost models. Using incremental checkpointing, Mona divides the application execution into a series of *checkpointing intervals*. Each checkpointing interval is further divided into two phases: partial checkpointing (during application execution) and final checkpointing. In partial checkpointing, we dynamically select the *cold* dirty pages that are less likely to be written by the application in the future and write them to PCM. In final checkpointing, we use a synchronous scheme and the application execution is temporarily suspended until the next checkpointing interval. With the dynamic partial checkpointing, Mona is able to reduce the downtime in the final checkpointing. Based on the access pattern of each dirty page, we develop a *coldness threshold* to identify the pages for partial checkpointing. When the elapsed time of a page since its last write access in DRAM is longer than the coldness threshold, the page becomes a candidate cold page.

To fully utilize the memory bandwidth, we propose to leverage the idle period during application execution to perform the partial checkpointing writes to PCM. However, those writes may pose interference to the application execution. Ideally, we should identify the lengthy idle periods that can accommodate the PCM writes. However, it is impossible or impractical to predict the length of the next idle period without a *prior* knowledge. Inspired by the previous study [31], we periodically estimate the idle period distribution and determine a *write threshold* for individual DRAM ranks according to the distribution. During the non-checkpointing period, only if the current idle period exceeds the write threshold, Mona performs a PCM write.

We further develop mechanisms of load balancing across DRAM ranks for both partial checkpointing and final checkpointing. In partial checkpointing, we consider the number of pages to write and the idle periods of all ranks, and periodically migrate the pages among them or tune the parameters (write threshold and coldness threshold) for each rank so that the partial checkpointing cost is balanced among DRAM ranks. In final checkpointing, we even up the number of dirty pages in each DRAM rank prior to perform PCM writes. Specifically, we perform page migration among DRAM ranks, because the cost of a page migration in DRAM is much lower than that of a PCM write.

To support the flexibility of different optimization goals, Mona develops a cost model to guide its decision, by estimating the checkpoint overhead or checkpointing interval. In this paper, we use two common scenarios as examples: first, minimize the checkpointing overhead given a checkpointing interval; second, minimize the checkpointing interval given a penalty budget on the application execution. Since these two scenarios are dual problems, this paper mainly focuses on the first scenario, and evaluates the second scenario in the experiment.

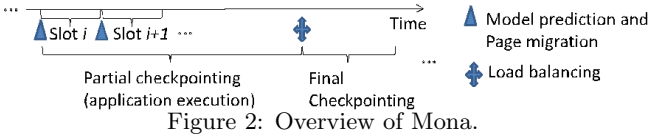


Figure 2: Overview of Mona.

Algorithm 1 The workflow of Mona for a checkpoint interval

```
1: /*Partial checkpointing*/
2: for each idle period e do
3:     Update histogram information;
4:     According to the write threshold configuration, Mona performs
       PCM writes with the dirty pages satisfying coldness threshold;/*Section 4.1.1*/
5: for each memory access r do
6:     if r is a write then
7:         Update page access pattern information;
8: for the beginning of a new slot do
9:     Perform model prediction and load balancing among DRAM
       ranks; /*Sections 4.1.2*/
10: /*Final checkpointing*/
11: Balance the dirty pages among DRAM ranks; /*Section 4.2*/
12: Write all the dirty pages to PCM;
```

The overall workflow of Mona on a checkpointing interval is given in Algorithm 1. In partial checkpointing, we use a histogram to represent the idle period distribution. For each idle period, we update the histogram. For each write request, we need to update access pattern information. The details of the update procedures will be presented in Sections 4.1.1 and 4.1.2. In final checkpointing, we first perform load balancing among DRAM ranks, and then write all the dirty pages to PCM. Note that our load balancing is guided by the cost model and, hence, it adapts to memory architectures and configurations. Afterwards, an incremental checkpoint is created and the next checkpointing interval starts. Figure 2 illustrates a checkpointing interval. We further divide a partial checkpointing period into multiple equal-sized *slots*. At the beginning of each slot, we perform the cost estimation and load balancing across DRAM ranks. The cost estimation is performed with a sliding history window, and the window length is equal to the checkpointing interval.

Finally, upon a system failure, Mona constructs the latest checkpoint and restarts the execution. In the event of failures during transferring a partial checkpoint from DRAM to PCM, we can choose to retry the checkpointing writes or adopt other existing failure recovery methods. Since the restarting process is not the main focus of this paper, Mona uses the classic recovery algorithm [23] for incremental checkpointing.

4. DESIGN AND IMPLEMENTATIONS

We describe the implementation details for key components in Mona, including the cost model and the load balancing module. The cost model guides the optimization for individual DRAM ranks, and load balancing aims at evening up the checkpointing cost of all DRAM ranks.

4.1 Cost Model

The cost model guides Mona to determine the threshold values of individual ranks including coldness threshold and write threshold under different optimization goals. As a case study, we mainly focus on the goal of minimizing the total checkpoint overhead given the checkpointing interval. Thus, in this section, we present the cost model of determining the threshold values under this optimization goal.

In a checkpointing interval, the total checkpoint overhead is the sum of the performance penalty caused by PCM writes in partial checkpointing and the cost of PCM writes in final checkpointing, denoted as $C_{partial}$ and C_{final}, respectively. The notations for the cost model are summarized in Table 1. Thus, the total checkpoint overhead is given by:

$$C_{total} = C_{partial} + C_{final}. \quad (1)$$

We further derive each cost component with the unit cost multiplying the number of PCM writes:

$$C_{total} = N_{partial} \times W_{partial} + N_{final} \times W_{PCM}. \quad (2)$$

In final checkpointing, the unit cost is simply the cost of a normal PCM write (W_{PCM}), which can be a constant obtained from specification or calibration. In partial checkpointing, Mona leverages the idle period to perform the PCM write. We present the detailed algorithm of determining the optimal $W_{partial}$ given the number of partial writes ($N_{partial}$) in Section 4.1.2.

Table 1: Notations in this paper

Notations	Description
R_{dram}	Number of DRAM ranks in Mona
R_{pcm}	Number of PCM ranks in Mona
C_{total}	Total checkpointing cost of a checkpointing interval
$C_{partial}/C_{final}$	Total cost of partial/final checkpointing
$N_{partial}$	Number of PCM writes in partial checkpointing
$W_{partial}$	Average cost of a PCM write in partial checkpointing
N_{final}	Number of PCM writes in final checkpointing
W_{PCM}	Average cost of a PCM write in final checkpointing
T_C	*Coldness threshold*
$\vec{t}$	*Write threshold* vector

Since we can determine the optimal $W_{partial}$ given $N_{partial}$, our task is to find $N_{partial}$ and N_{final} so that C_{total} is minimized. There is a tradeoff between $N_{partial}$ and N_{final}. If more pages are written to PCM during partial checkpointing ($N_{partial}$ is larger), N_{final} tends to be smaller. On the other hand, some pages may be written to PCM for multiple times in one checkpointing interval. Thus, we should determine a suitable coldness threshold to ensure the effectiveness of partial checkpointing. Given a coldness threshold T_C, we develop the algorithm to estimate $N_{partial}$ and N_{final} (described in Section 4.1.1).

To put them all together, the cost model determines the threshold values in Algorithm 2. We iterate all the possible values for T_C, and find the best C_{total}. To reduce the number of iterations, we iterate T_C with a unit of δ cycles (δ is set to be 1% of the checkpointing interval in our experiments). Additionally, in practice, we observe that C_{total} is usually a concave function to T_C. Thus, we can stop the iterations right after reaching the minimum T_C.

Algorithm 2 The workflow of determining suitable threshold values with the cost model

1: Initialize $curCost = 0$, $C_{total} = +\infty$, I to be the length of the checkpointing interval;
2: **for** $T_C = 0, \delta, 2\delta, ..., I$ **do**
3: According to T_C, determine $N_{partial}$ and N_{final}; /*Section 4.1.1*/
4: According to $N_{partial}$ in Line 3, determine $W_{partial}$; /*Section 4.1.2*/
5: Based on the results of Lines 3 and 4, calculate the total checkpointing cost to be $curCost$;
6: **if** $curCost < C_{total}$ **then**
7: $C_{total} = curCost$;
8: Update threshold values with the new obtained setting;
9: Return threshold values that result in the smallest C_{total};

4.1.1 Estimating $N_{partial}$ and N_{final}

We now discuss how to estimate $N_{partial}$ and N_{final}, given a coldness threshold T_C. The estimation is based on a history window with length equal to one checkpointing interval. We derive $N_{partial}$ and N_{final} based on the historical memory access statistics. Specifically, the coldness of a page is indicated by the elapsed time since its last write access. We record each page's first write access time, last write access time and the total number of write accesses, denoted as T_{first}, T_{last} and N_{write}, respectively.

We model the problem as a sweeping line on a plane. An example is given in Figure 3. Each page P_i is represented as an interval or a dot $<T_{first}, T_{last}, addr>$ on the plane, where $addr$ is the starting address of P_i (such as P_1 through P_4 in Figure 3). Assuming line l at time T_l is sweeping from the plane's right boundary to its left boundary. Each time line l sweeps a distance of δ. The time difference between l and the right boundary is thus viewed as the *coldness threshold* T_C. That is, as line l sweeps to the left, we essentially increase T_C as in Line 2 of Algorithm 2. Depending on a page's access span $T_{span} = T_{last} - T_{first}$ and its N_{write}, the sweeping line l divides all the pages into the following three categories:

- *Category 1:* If P_i's T_{first} is larger than T_l (e.g., P_4), this page is not cold and should be written to PCM only once in final checkpointing.
- *Category 2:* If P_i's T_{last} is smaller than T_l (e.g., P_1), it has been written to PCM in partial checkpointing. To predict the number of writes to PCM, we further compare its T_{span} and N_{write} in two cases: 1) If P_i's N_{write} is larger than $\lceil T_{span}/T_C \rceil$, P_i is frequently written. We will only write it once after its last write access, since P_i will never be cold enough during its T_{span}. 2) Otherwise, P_i will be written to PCM at most N_{write} times under the assumption that each of the time span between two successive write accesses is long enough that P_i has to be written once.
- *Category 3:* For P_i's T_{span} is stabbed by line l (e.g., P_2 and P_3), we apply the same idea as in the second category: 1) If P_i's N_{write} is larger than $\lceil T_{span}/T_C \rceil$, we will only write it once during final checkpointing. 2) Otherwise, P_i will be written at least once. Moreover, it may also be written at most $N_{write} - 1$ times during partial checkpointing.

Given the three categories, we calculate $N_{partial}$ and N_{final} as in Algorithm 3 by counting all the pages in each category.

The estimation is light-weight. For each page, Mona maintains three metadata (T_{start}, T_{end} and N_{access}) in the form of three integers. They are kept in the page-header,

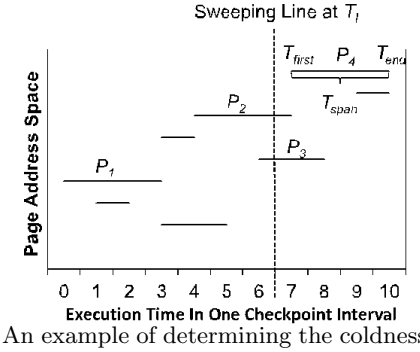


Figure 3: An example of determining the coldness threshold

Algorithm 3 Finding $N_{partial}$ and N_{final} given a coldness threshold T_C at sweeping line T_l

```
1: Initialize N_partial = 0, N_final = 0;
2: for each page P_i do
3:    if T_first > T_l then
4:       N_final ++;
5:    else if T_last < T_l then
6:       if N_write > ⌈T_span/T_C⌉ then
7:          N_partial ++;
8:       else
9:          N_partial += N_write;
10:   else
11:      N_final ++;
12:      if N_write <= ⌈T_span/T_C⌉ then
13:         N_partial += N_write − 1;
```

which requires no hardware cost. The maintenance can be piggy-backed into the page-header update process of the virtual-memory system of the operating system. We adopt a similar approach proposed by Zhou et al. [38]. After each DRAM write operation, we update the coldness information. Noted that this update cost is very small, as we only need to update one piece of information.

4.1.2 Estimating $W_{partial}$

After getting $N_{partial}$ and N_{final}, we now estimate the average cost of a PCM write ($W_{partial}$) during partial checkpointing. Recall that, during partial checkpointing, we take advantage of idle periods to perform PCM writes. Thus, the actual cost of a PCM write on the application execution is affected by the length of an idle period. If the idle period is too short, the PCM write can cause significant penalty, because the PCM write is much slower than the DRAM access. On the other hand, if the idle period is lengthy, there are opportunities of performing multiple PCM writes. However, it is impossible to predict the length of the current idle period without a priori knowledge. Thus, we estimate the idle period length distribution in a slot. Then, we adopt a simple approach to determine whether to perform a PCM write: if the current idle period is longer than the *write threshold*, we perform the PCM write.

We use a histogram to represent the distribution of idle lengths. Particularly, the histogram is used to hold the frequency counts of the idle period lengths. Suppose the histogram has $T+1$ buckets denoted as $Hist[i]$, $i = 0, 1, ..., T$, where $Hist[i]$ denotes the frequency count for the idle period with a length of i cycles. $Hist[0]$ simply represents the number of memory access cycles. Given a write threshold of t cycles, a memory access delay of $(t + W_{PCM} - i)$ will happen for an idle period of i cycles.

To support multiple PCM writes in a lengthy idle period, we use a vector $\vec{t}$ to represent the write threshold values

Algorithm 4 Obtain the optimal write threshold vector $\vec{t}$ for an idle period

```
1: Initialize t⃗ = null, curWrite = 0, curCost = 0, minCost = +∞;
2: FindTI(t, curWrite, curCost, minCost, Hist);
Procedure FindTI (t⃗, curWrite, curCost, minCost, Hist)
1: for t = 0, 1, ..., T do
2:    for i = t + 1, 2, ..., t + W_PCM do
3:       curCost+ = (t + W_PCM − i) × Hist[i];
4:       curWrite+ = Hist[i];
5:       if curWrite ≥ N_partial then
6:          break;
7:       else
8:          if i = t + W_PCM then
9:             for i = t + W_PCM, ..., T do
10:               Hist′[i]=Hist[i − (t + W_PCM)];
11:               FindTI(t⃗, curWrite, curCost, minCost, Hist′);
12:   if curCost < minCost then
13:      minCost = curCost
14:      T_I = t;
15:   else
16:      add T_I to t⃗;
```

within the same idle period. $\vec{t}[i]$ is the write threshold value for the ith PCM write. Consider the scenario that an idle period starts. We perform the first PCM write if the idle period reaches $\vec{t}[0]$ cycle. After the PCM write finishes, if we are still in the same idle period, we will perform the second PCM write if the idle period lasts for $\vec{t}[1]$ cycles more. Thus, we perform a recursive algorithm to find the vector, as illustrated in Algorithm 4. When the number of PCM writes that have been performed is smaller than $N_{partial}$, we need to find one more level of PCM writes in the relatively long idle periods. We derive this process as a subproblem of the original problem (Line 11), with the updated histogram as input to Procedure *FindTI*.

Previous studies [13, 31, 17] have shown that the idle periods usually demonstrate a "long tail" phenomenon, where there are many short idle periods and very few long idle periods. Thus, as we increase the write threshold value t, the optimal cost first decreases as we can skip the short idle periods, and then increases as we miss those sufficiently long idle periods. We take advantage of this property to reduce the computation for Procedure *FindTI* (Lines 12–16).

4.2 Load Balancing

The cost model guides the configuration parameters for an individual DRAM rank. However, the total page migration time is bounded by the longest migration time among all DRAM ranks. There are two kinds of skews among DRAM ranks. First, in partial checkpointing, a DRAM rank may consist of many dirty pages and too few lengthy idle periods. This causes the imbalance in the cost of partial checkpointing. Second, in final checkpointing, a DRAM rank may consist of a much larger number of dirty pages than other ranks. This causes the imbalance in final checkpointing. In the following, we present algorithms for addressing the cost imbalance caused by these two kinds of skews.

Balancing final checkpointing. In final checkpointing, the set of dirty pages to be written from DRAM to PCM is already known. The goal is to balance the dirty pages among DRAM ranks. Since DRAM accesses are much faster than PCM writes, one simple approach is to perform page migrations among DRAM ranks before starting PCM writes. The page migration moves the dirty pages from the DRAM

rank that has more dirty pages (denoted as H) to another DRAM rank with fewer dirty pages (denoted as L). If L does not have sufficient free space for the migrated dirty pages, we move clean pages from L to H. Particularly, we choose a dirty page from H to L and randomly choose a clean page from L to H as appropriate. We further adopt the parallel page migration approach of the previous study [31] to better utilize the memory bandwidth. The migration ends when the dirty pages are even among the DRAM ranks.

Balancing partial checkpointing. We have developed two approaches to balance partial checkpointing: one with page migrations and the other with parameter tuning (namely LB-PM and LB-PT, respectively). With the cost model, Mona is able to compare the estimated cost of these two approaches, and choose the one with the smaller cost.

In LB-PM, we attempt to balance the partial checkpoint overhead among DRAM ranks through page migrations. Here, we only need to balance the partial checkpointing cost, since the final checkpointing cost among ranks will be balanced later. The basic idea is to migrate pages from the rank with the highest cost to the rank with the lowest cost.

The page migration is performed in an iterative manner. In each iteration, we first run the cost model and get the total cost of partial checkpointing for each rank. If the cost is almost even among DRAM ranks (the difference is smaller than 3% in our experiments), we end the page migration process. Otherwise, we adopt a greedy method to choose the rank with the highest cost as $Rank_S$ and the rank with the lowest delay as $Rank_T$. We migrate pages from $Rank_S$ to $Rank_T$. The number of pages migrated at a time is a tuning parameter, which affects the overhead of running the cost model and the evenness of the migration cost among DRAM ranks. We have experimentally evaluated its impact and choose eight pages at a time as the default setting.

The page chosen to be migrated needs careful consideration. To reduce the number of page migrations, we should choose those pages that affect the checkpointing cost more significantly. Thus, we choose the dirty pages with the least write localities in $Rank_S$. In LRU, we choose the page from the least recent accessed position in the LRU queue. If $Rank_T$ does not have sufficient free space, we choose the clean pages with the least read localities in $Rank_T$ and move them to $Rank_S$. Since the selected pages are the least recent accessed ones, we ignore their impact on the histogram of idle periods for simplicity.

Unlike LB-PM resulting in the page migration overhead, LB-PT simply adjusts the *write threshold* values and *coldness threshold* to balance the cost of partial checkpointing. Specifically, through parameter tuning on T_C and $\vec{t}$, we adjust the $C_{partial}$ and C_{final} for each rank so that all ranks have similar $C_{partial}$. Meanwhile, the largest total checkpointing cost of all ranks is minimized. Please note, LB-PT considers the load balancing for final checkpointing as well. LB-PT relies on the intermediate output in the cost model (Algorithm 2) to find the threshold values when adjusting $C_{partial}$ and C_{final}. Particularly, the computation process of the cost model gives different mappings from (T_C, $\vec{t}$) to ($C_{partial}$, C_{final}). Originally, we choose the lowest total cost for each rank. Now, we need to choose T_C and $\vec{t}$ that can balance the cost across different ranks.

We briefly describe the load balancing algorithm of LB-PT as follows. For Rank i ($1 \leq i \leq R_{dram}$), we maintain a sorted queue Q_i to hold all pairs of $C_{partial}$ and C_{final}, in the ascending order of $C_{partial}$. Initially, we pop the head of each Q_i and form a set of possible cost combinations, $Set_{candidate}$. We calculate the total cost among all ranks (denoted as $\mathfrak{C}$) after balancing final checkpointing. After that, we remove one pair of $C_{partial}$ and C_{final} with the highest $C_{partial}$ from $Set_{candidate}$ and add a new pair of $C_{partial}$ and C_{final} from the corresponding queue. We repeat this process until we find the minimal $\mathfrak{C}$. Then, we can find the corresponding thresholds for each rank based on $Set_{candidate}$.

4.3 Other Implementation Details

Mona adds a few new components to the memory controller and operating system. Our design and implementations balance the issues on simplicity and efficiency. For example, the memory requests from page migrations are designed off the critical path of memory accesses, giving the priority to the memory accesses from other logics. In the following, we briefly present the following issues for Mona.

Multi-level checkpointing. We extend Mona to handle multi-level checkpointing in two steps, following the design of previous NVRAM-assisted multi-level checkpoint designs [6, 7]. First, each node performs local checkpointing according to Mona. Second, after several local checkpoint intervals, a global checkpoint is initiated. A new global checkpoint is created from the latest local checkpoints and stored in the parallel file system.

Runtime and storage overhead. The structure complexity and storage overhead of Mona are similar to the previous proposals, e.g., [25, 31]. For example, our design has small DRAM space requirement (less than 2% of the total amount of DRAM), and an acceptable runtime overhead in maintaining the cost model and page access statistics (less than 3% of the total application execution time). For page migrations, only page descriptors and their mapping tables are stored. For the histogram in the cost model, we use the two-array storage method of the previous study [31] to reduce the storage overhead.

Limitations. Compared with existing NVRAM-assisted checkpoint designs, Mona has the following limitations. First, Mona relies on the history of memory accesses for cost estimation. More advanced workload prediction mechanisms are helpful to improve the robustness. Second, Mona has more PCM writes, which decrease PCM's lifetime. Our on-going work is to integrate/adapt the wear-leveling techniques to extend the lifetime of PCM (e.g., [24, 4]). Also, another possible solution is to leverage write consolidation techniques [32] to reduce the number of writes. We leave it as future work. Nevertheless, the predictions [25] for future PCM cells suggest significant endurance improvement (from 10^9 in 2012 to 10^{15} writes by 2022).

5. EVALUATION

5.1 Experiment Setup

We have developed a cycle-accurate simulator based on PTLSim v3.0 simulator [36]. Our simulator models all relevant aspects of the OS, memory controller, DRAM and PCM devices. In the simulator, the DRAM can be configured with different capacities (64GB to 512GB). For a fair comparison, we adopt the configurations from the previous study [6]. We simulate multiple CPUs, each running at 2.667 GHz, L1 I/D cache of 48 KB each and L2/L3 shared cache 256KB/4MB. The cache line and page size are set at 64B and 4KB, respectively. We use the DDR3-

Table 2: Mixed workload: memory footprint (FP), memory accesses statistics per 5×10^8 cycles ($Mean$ and $\frac{Stdev}{Mean}$).

Name	FP (MB)	$Mean$ (10^6)	$\frac{Stdev}{Mean}$	Applications
M1	661	0.6	1.02	gromacs, gobmk, hmmer, bzip
M2	1477	1.7	1.11	bzip, soplex, sjeng, cactusADM
M3	626	2.9	0.59	soplex, sjeng, gcc, zeusmp

1333 architecture from Micron's configuration[1]. By default, we assume a 16-rank hybrid memory system, with 128GB DRAM in 8 homogeneous ranks and with 256GB PCM in another 8 homogeneous ranks. The PCM can be configured with different PCM write delays. By default, we assume the write delay of PCM is 10 times of that of DRAM. The average latencies of DRAM and PCM are 100 and 1000 cycles, respectively.

Workloads. Since it is hard to mimic the memory trace of a real large-scale parallel processing system, we use both micro and macro benchmarks for performance evaluation. Micro benchmarks are used to perform detailed evaluation on using Mona as a local checkpoint design of each machine in a large-scale system. In contrast, macro benchmarks are used to assess multi-level checkpointing in a parallel processing system. Particularly, we execute the NAS Parallel Benchmark (NPB) [29] on our local cluster.

Micro benchmark. The micro benchmark include real workloads as well as synthetic workloads on a single machine. For real workloads, we selected 12 applications from SPEC 2006. These applications have widely different memory footprints and localities. To evaluate our techniques under the context of multi-core CPUs, we study 3 mixed workloads (denoted by M1, M2, and M3), each with four different applications (see Table 2) as one batch. Each mixed workload is generated by starting the four applications at the same time as one batch. In order to scale the memory footprint and accesses, we run 50 batches of each workload simultaneously in the system. The memory access statistics of these workloads, including the average number of memory accesses ($Mean$) and the standard deviation ($\frac{Stdev}{Mean}$) per 5×10^8 cycles, is shown in Table 2. We report the results of three particular applications, i.e., cactusADM, gcc, and gemsFDTD (denoted by S1, S2, and S3, respectively). Also, we run 50 instances each of the workload for scaling the memory footprint and memory accesses. We set the number of CPU cores according to the number of applications.

Synthetic workloads allow us to extensively evaluate different memory access patterns. For synthetic workloads, we assume that the arrival of page accesses follows a Poisson distribution, and thus the idle periods follow an exponential distribution with an expected value σ. The default value of σ is set at 2,000 cycles. We assume that the accesses over different pages follow a Zipf distribution with 1.5 as the skewness parameter. The read-write ratio is set at 3:1.

For each micro benchmark workload, we measure the performance after system warm-up (i.e., DRAM is filled up). For Mona, the initial *coldness threshold* and *write threshold* are derived based on the system statistics collected during the warm-up period. Each measurement lasts for 15×10^9 cycles in the original PTLSim simulation, which represents a stable and sufficiently long execution behavior [31]. The default checkpointing interval is 1×10^9 cycles, and the prediction slot is 2×10^8 cycles. Such fine-grained checkpointing intervals are specifically designed for the rigid requirement of critical jobs. We observed similar performance improvement when varying the interval and slot in a reasonable domain.

[1]http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb DDR3 SDRAM.pdf

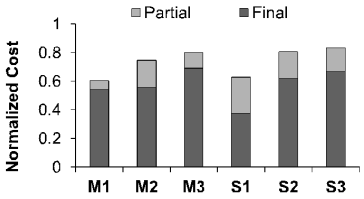


Figure 4: Performance of real workloads

Marco benchmark. We run NPB [29] in a local cluster. NPB includes eight benchmark programs derived from computational fluid dynamics (CFD) applications. We run both class B and class C for each of the NPB programs. The number of processes is 64 by default. We focus on the results in class C. The cluster has 16 nodes, each with CentOS 5.0, two Intel Xeon E5430 4-core processors and 24GB DRAM. They are interconnected via QDR InfiniBand network. NFS is used as the shared file system. The MPI library is Intel MPI 4.0.1. We adopt a hybrid approach of simulation/execution: the local checkpointing process is simulated by our simulator, and the other operations like MPI communications and global checkpointing are performed on the real environments. We focus on total checkpoint overhead by summing all local checkpoint overhead from each simulator.

Comparisons. We compare the state-of-the-art NVRAM-assisted checkpointing method [6, 7] (denoted as **baseline**) with Mona. In the baseline method, the system is suspended and then incremental checkpointing is performed. Its cost is the elapsed time in writing all dirty pages from memory ranks to PCM. To minimize the elapsed time in concurrent writing, like Mona, we also perform page migrations so as to even out the write workloads across different ranks. For clarity of presentation, we normalize the partial and final checkpointing cost of Mona to the total cost of the baseline method.

5.2 Overall Micro Benchmark Results

We first present the overall comparison results in Figure 4, where the cost of Mona is normalized by that of the baseline checkpointing method. For real workloads, Mona achieves 16% to 39% of performance improvement against the baseline method on the overall checkpointing cost. Mona can adjust the costs between partial and final checkpointings for different workloads. In Figure 5, we plot the results of synthetic workloads. We vary the PCM write delay per page with 500, 1000 (default), and 2000 cycles (denoted as PCM1, PCM2, and PCM3, respectively). We also vary the expected idle period with 1000, 2000 (default), and 4000 cycles (denoted as Idle1, Idle2, and Idle3, respectively). Mona achieves an improvement of up to 65% over the baseline method. The improvement is relatively higher for short PCM write delays and long idle periods.

To further understand the improvement, we plot the actual number of PCM writes of the synthetic workloads in Figure 6. For Mona, we show the counts for three kinds of PCM writes: PCM writes in final checkpointing, PCM writes with and without causing a delay during partial checkpointing. Mona has a large portion of PCM writes in

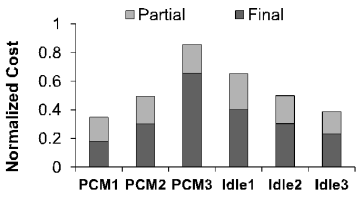


Figure 5: Performance of synthetic workloads

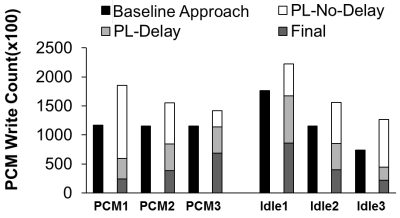


Figure 6: Number of PCM writes in synthetic workloads

partial checkpointing that cause no penalty under a short PCM write delay or a long idle period. That suggests our cost model is able to adapt to the system parameters and reduce the cost of partial checkpointing. Although Mona performs more partial checkpointing, the overall cost still becomes less. This phenomenon can also be observed by comparing the cost ratio of partial checkpointing to total checkpointing. Under the shortest PCM write delay, the ratio of partial checkpointing increases to about 47% of the total cost. On average, Mona has 28% more PCM writes than the baseline approach on all workloads.

To better understand the performance of Mona, in the following sections, we show more detailed results with the synthetic workloads (PCM2 by default).

5.3 Cost Model Evaluation

To reveal the effectiveness of our cost model, we present the prediction results for a memory rank during one checkpointing interval. In Figure 7, we plot the predicted write counts of partial and final checkpointings for DRAM Rank 1, under different coldness thresholds. We observe similar results on other DRAM ranks. With a larger threshold, more pages are written during final checkpointing, since they are not "cold" enough to be written to PCM during partial checkpointing. Thus, the dominant portion of the total cost changes from partial checkpointing to final checkpointing. As a result, the total cost decreases at first, and then increases as the threshold continues to enlarge. Through this cost model, we can find the minimal cost and its corresponding *coldness threshold* and *write threshold*.

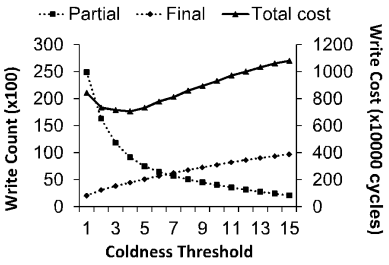
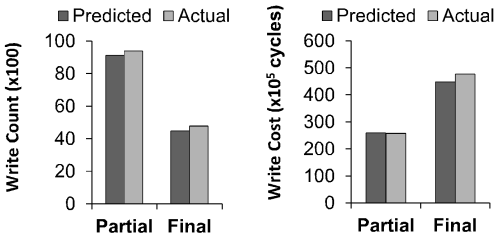
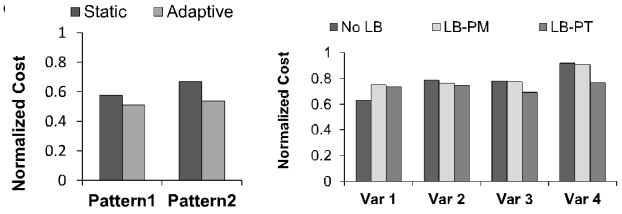


Figure 7: Number of PCM writes and cost estimation (on PCM2)



(a) Write counts (b) Write costs

Figure 8: Comparison between estimation and actual



(a) (b)

Figure 9: Individual techniques: a) adaptive cost estimation, b) load-balancing methods.

Figure 8 validates our prediction by comparing the predicted and the actual write counts and costs. The error is only at most 6% for both predictions. We observed similar results on other DRAM ranks. This implies that our model can guide the checkpointing process with an accurately predicted cost.

5.4 Impacts of Individual Techniques

Adaptive Cost Estimation. We now compare the results between dynamic and static predictions in Figure 9a. For the dynamic method, we keep a sliding window and tune the system parameters at every slot. For the static method, we only use the initial predicted thresholds. We show the results on two dynamic page access patterns: (Pattern 1) $w_i = 0.95w_0 \cdot (i \bmod 2) + w_0 \cdot ((i+1) \bmod 2)$ and (Pattern 2) $w_i = w_0(1 + i\%)$, where w_i is the write ratio within the $(i+1)$-th checkpoint, and w_0 is the default setting (0.25). Pattern 1 simulates the case where the workload periodically changes. Pattern 2 simulates the case where write requests gradually become dominant in the workload. As shown in Figure 9a, our adaptive prediction achieves 6% to 13% better performance than the static method. A higher improvement is observed for Pattern 2, because there are more write requests in Pattern 2, incurring more performance penalty in the static cost model.

Load balancing methods. To investigate the performance advantage of load balancing, we compare three approaches in Figure 9b: Mona without load balancing, the page-migration method, and the parameter-tuning method. We set the variations of mean idle period and read-write ratio between different ranks as 200 and 0.1 (Var1), 400 and 0.2 (Var2), 600 and 0.3 (Var3), to 800 and 0.4 (Var4). Under small variations, both migration methods do not perform well, because the migration cost or the parameter tuning cost may exceed the benefits on the overall checkpointing process. Under large variations, both methods outperform the method without any load balancing. Specifically, the parameter-tuning method achieves the best result and outperforms the page-migration method by about 10% with the largest variation. This is because the page-migration method involves a relatively high migration overhead. For

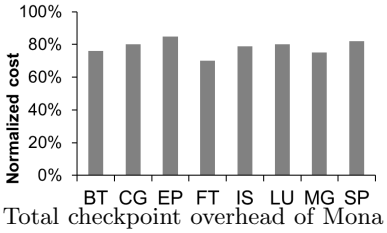


Figure 10: Total checkpoint overhead of Mona (normalized to the baseline approach). Local and global checkpointing intervals are 40 and 400 seconds, respectively.

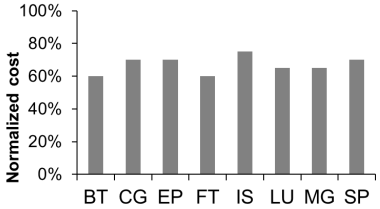


Figure 11: Total checkpoint overhead of Mona (normalized to the baseline approach). Local and global checkpointing intervals are 1 and 10 seconds, respectively.

the parameter-tuning method, the overhead of not choosing the optimal settings for individual ranks is worthy to achieve a better overall performance. In experiments, Mona chooses the best policy to be the one with the smallest estimated cost (Section 4.2).

5.5 Macro Benchmark Results

We now present the macro benchmark results on multi-level checkpointing. Overall, we have observed that Mona significantly improves the performance of multi-level checkpointing, thanks to the efficiency in the local checkpointing of each machine. More frequent checkpoints increase system availability, but degrades the baseline approach significantly.

Figure 10 reports the normalized total checkpoint overhead on coarse-grained checkpointing when the local and global checkpointing intervals are 40 and 400 seconds, respectively. The average checkpoint overhead reduction of Mona is 22% over the baseline approach. The performance improvement of Mona is slightly smaller for coarse-grained checkpointing than those reported in the paper, since some pages need to be written for multiple times.

Figure 11 shows the comparison with fine-grained checkpointing when the local and global checkpointing intervals are 1 and 10 seconds, respectively. The average checkpoint overhead reduction of Mona is 33% over the baseline approach. The improvement is much more significant than those observed in Figure 10. Mona achieves a much lower overhead while also delivering fine-grained checkpointing for critical jobs. The findings with the hybrid local/global checkpointing algorithm are consistent with the results in a single node.

5.6 Sensitivity Studies

We present our sensitivity studies in Figure 12. First, we vary the DRAM size to be 64GB, 128GB (default), 256GB, and 512GB. A higher performance improvement is observed as the DRAM size increases. Second, we vary the number of DRAM ranks to be 2, 4 and 8 (default), and keep the number of DRAM ranks equal to the number of PCM ranks. The performance improvement slightly improves for more ranks

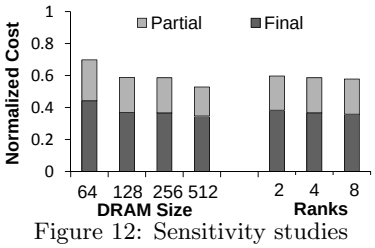


Figure 12: Sensitivity studies

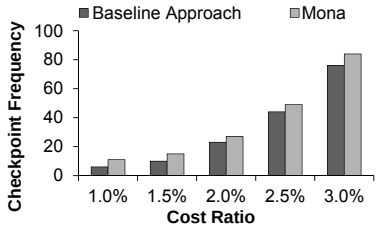


Figure 13: Results on optimizing checkpoint interval

in the memory system. We also vary the read ratio from 0.6 to 0.9, and have observed similar performance improvement.

5.7 Optimizing Checkpointing Interval

Finally, we present the results when Mona's optimization goal is set to minimize the checkpoint interval. This goal also means minimizing the restart time (i.e., the total time of loading the lastest checkpoint and re-execution till the point of failure). We vary the ratio of total allowed cost budget from 1% to 3% to the application execution time and compare the total checkpointing count. As shown in Figure 13, Mona achieves 10% to 80% of improvement on the checkpoint frequency. The improvement generally decreases as the allowed cost ratio increases. This is because as the interval becomes smaller, Mona tends to behave like the baseline method. In the extreme case where the allowed cost ratio is very large, Mona and the baseline method will have the same checkpointing cost, which writes out every page right after it becomes dirty.

6. RELATED WORK

There have been decades of research on checkpointing-restart mechanisms. More related works can be found in a recent survey [9]. Incremental checkpointing [1] can reduce the checkpoint file size, and speculative checkpointing [18] can reduce the downtime. Mona adopts page-level incremental checkpointing, and the partial checkpointing of Mona is inspired by speculative checkpointing. Differently, our checkpointing leverages idle period distributions, and relies on a cost model to minimize the interference to the application execution.

Similar to our study, Dong et al. [6] proposed a hybrid checkpoint model by investigating different architectural design options for PCM-based checkpointing systems. Zhou et al. [37] used writeback information to partition bandwidth for hybrid memory. Essen et al. [30] developed PerMA NVRAM simulator to study different roles of NVRAM in data-intensive architectures. Li et al. [16] studied the opportunities of NVRAM for extreme-scale scientifc applications. Our study further enables the real-time checkpointing for extreme-scale applications. Mona addresses the limitation of the previous work [6, 7, 14] with two novel techniques. First, Mona has a model-guided dynamic partial checkpointing during application execution, which can accurately write back the more valuable pages. Second, Mona develops finer-grained load balancing mechanisms among memory ranks.

NVRAM has been used for checkpointing in other scenarios. UniFI [26] aims at energy saving and fault tolerance by exploiting NVRAM in both CPU caches and main memory. Yoon et at. [35] proposed a new architecture for Multilevel-cell phase change memory to cope with the problem of resistance drift. However, this work focus on the hardware design, instead of new checkpoint system. In [21], Oikawa proposed a checkpointing mechanism for operating system kernel rejuvenation. In our study, we target at a hybrid memory system, since the low access latency of PCM make it infeasible to independently serve as the main memory.

As for multi-level checkpointing, Moody et al. [19] performed extensive modeling and evaluations and showed that the benefits of multi-level checkpointing increase as the system size increases. Sato et al. [27] studied a non-blocking checkpoint system to reduce the I/O contention on PFS. This paper demonstrates that a novel NVRAM-assisted design can further improve multi-level checkpointing.

7. CONCLUSION

Real-time in-memory checkpointing is essential for reliable large-scale parallel processing systems. This paper optimizes the runtime overhead of in-memory checkpointing in a hybrid memory system with PCM and DRAM. It embraces dynamic partial checkpointing during application execution with optimizations on reducing the interference and load balancing among ranks to reduce the checkpoint overhead. Those decisions are guided by a cost model. We conduct simulation-based experiments to evaluate the performance of Mona under various workloads. Compared to the state-of-the-art checkpointing method, Mona reduces the total checkpointing cost by up to 65% under the goal of minimizing the checkpointing cost.

8. ACKNOWLEDGMENTS

Shen's work was done when he was a visiting student at NTU Singapore. This work is supported by a MoE AcRF Tier 2 grant (MOE2012-T2-1-126) in Singapore.

9. REFERENCES

[1] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS*, 2004.

[2] K. Bailey and et al. Operating system implications of fast, cheap, non-volatile memory. In *HotOS*, 2011.

[3] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 2005.

[4] C.-H. Chen and et al. Age-based PCM wear leveling with nearly zero search cost. In *DAC*, 2012.

[5] J. Condit and et al. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[6] X. Dong and et al. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *SC*, 2009.

[7] X. Dong and et al. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM TACO*, 2011.

[8] Y. Du and et al. Delta-compressed caching for overcoming the write bandwidth limitation of hybrid main memory. *ACM TACO*, 2013.

[9] I. Egwutuoha and et al. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 2013.

[10] E. N. Elnozahy and et al. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE TDSC*, 2004.

[11] E. N. M. Elnozahy and et al. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 2002.

[12] S. Gao, J. Xu, B. He, B. Choi, and H. Hu. Pcmlogging: Reducing transaction logging overhead with pcm. CIKM, 2011.

[13] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. Improving energy efficiency by making dram less randomly accessed. In *ISLPED*, 2005.

[14] S. Kannan and et al. Optimizing checkpoints using nvm as virtual memory. In *IPDPS*, 2013.

[15] B. C. Lee and et al. Phase-change technology and the future of main memory. *IEEE Micro*, 2010.

[16] D. Li and et al. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *IPDPS*, 2012.

[17] K. T. Malladi and et al. Towards energy-proportional datacenter memory with mobile dram. In *ISCA*, 2012.

[18] S. Matsuoka and et al. Speculative checkpointing: Exploiting temporal affinity of memory operations. In *HPC Asia*, 2009.

[19] A. Moody and et al. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*, 2010.

[20] D. Narayanan and O. Hodson. Whole-system persistence with non-volatile memories. In *ASPLOS*. ACM, March 2012.

[21] S. Oikawa. Independent kernel/process checkpointing on non-volatile main memory for quick kernel rejuvenation. In *ARCS*, 2014.

[22] F. Petrini. Scaling to thousands of processors with buffered coscheduling. In *Scaling to New Heights Workshop*, 2002.

[23] J. S. Plank and et al. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, 1995.

[24] M. K. Qureshi and et al. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *MICRO*, 2009.

[25] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *ICS*, 2011.

[26] S. Sardashti and et al. UniFI: leveraging non-volatile memories for a unified fault tolerance and idle power management technique. In *ICS*, 2012.

[27] K. Sato and et al. Design and modeling of a non-blocking checkpointing system. In *SC*, 2012.

[28] B. Schroeder and et al. Dram errors in the wild: a large-scale field study. In *SIGMETRICS*, 2009.

[29] The NAS Parallel Benchmarks. http://www.nas.nasa.gov/resources/software/npb.html, 2013.

[30] B. Van Essen and et al. On the role of nvram in data-intensive architectures: An evaluation. In *IPDPS*, 2012.

[31] D. Wu and et al. RAMZzz: rank-aware dram power management with dynamic migrations and demotions. In *SC*, 2012.

[32] F. Xia and et al. Dwc: Dynamic write consolidation for phase change memory systems. In *ICS*, 2014.

[33] J. Xie, X. Dong, and Y. Xie. 3D memory stacking for fast checkpointing/restore applications. In *IEEE 3DIC*, 2010.

[34] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. USENIX FAST, 2015.

[35] D. H. Yoon and et al. Practical nonvolatile multilevel-cell phase change memory. SC, 2013.

[36] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS*, 2007.

[37] M. Zhou and et al. Writeback-aware bandwidth partitioning for multi-core systems with pcm. In *PACT*, 2013.

[38] P. Zhou and et al. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS XI*, 2004.