Balanced Parity Update Algorithm with Queueing Length Awareness for RAID Arrays

Youxu Chen[1,2], Yinlong Xu[1,3], Yongkun Li[1,2], Jun Xu[4]
1. School of Computer Science and Technology, University of Science and Technology of China
2. AnHui Province Key Laboratory of High Performance Computing, Hefei, China
3. Collaborative Innovation Center of High Performance Computing, National University of Defense Technology
4. Shannon Laboratory, Central Research Institute, Huawei Technologies Co., Ltd., China
Email: cyx1227@mail.ustc.edu.cn, {ylxu, ykli}@ustc.edu.cn, xujun09@huawei.com

***Abstract*—In parity-based RAID arrays, to update a data chunk, the corresponding parity chunk(s) must be updated accordingly so as to keep data consistency and availability. To achieve this, either read-modify-write (RMW) or read-construct-write (RCW) could be used. Traditional parity update algorithm always selects the one requiring fewer pre-reads so as to reduce the total number of I/Os, but it may aggravate the skewness of I/O queues on disks, and thus degrades the system performance.**

In this paper, we propose a balanced parity update algorithm with queueing length awareness, BPU, which takes the number of pre-reads, the skewness of I/O queues on disks, and real-time workload into consideration when selecting RCW or RMW to update parity chunks. We implement a prototype system with BPU to evaluate its performance. Experimental results show that the length of I/O queues on disks in a RAID array may be highly skewed when using traditional parity update algorithm, and thus severely degrades the system performance. With BPU, we can reduce the average response time by up to 10%. We also study the performance of BPU under different system configurations, and provide multiple insights for adjusting the parameters of BPU so as to optimize its performance.

***Index Terms*—Parity Update; RAID; Queue; Skewness; Performance**

I. Introduction

Due to the development of data collecting devices, massive data are produced from many areas, such as scientific research, engineering applications and people's daily life. The fast growth of data size makes a single storage device be hard to provide enough storage capacity and I/O bandwidth as required. *Redundant Arrays of Inexpensive Disks* (RAID) [11, 6] provides a cost-effective approach to meet the requirement of large storage space and fault tolerance with commercial storage devices, such as traditional *hard-disk drives* (HDDs) and flash-based *solid-state drives* (SSDs) [3, 5].

Along with the increase of data volume, modern storage systems may use hundreds or thousands of inexpensive disks. As a result, system component failure like device failure becomes a very common behavior [12, 15]. To prevent data loss, data redundancy is introduced into large scale storage systems. *N*-way replication [17] and erasure code [8, 13] are two common approaches to provide data redundancy. With *N*-way replication, a data chunk is replicated *N* copies and stored in *N* different disks. For an (n, r) erasure code, r data chunks are encoded into n encoded chunks such that all data chunks can be decoded from any m $(r \leq m < n)$ ones of the encoded chunks, and so the erasure code prevents the loss of any $n-m$ encoded chunks.

Erasure codes can provide the same data availability with *N*-way replication, while only incurs an order of magnitude smaller storage overhead, so they are often deployed in RAID arrays. The commonly used erasure codes are *systematic* and *maximum distance separable* (MDS). Examples of erasure codes include RAID-5 against single failure; RDP [7], EVEN-ODD [4], and X-code [18] for RAID-6 against double failures, as well as Cauchy Reed-Solomon (CRS) [14] codes against any number of disk failures.

In a systematic and MDS code, apart from r data chucks, other $n-r$ encoded chunks are called parity chunks. Each of the parity chunks is the XOR-sum of some data chunks. To keep data consistency and data availability, the parity chunks need to be updated when any of the data chunks that are used to generate the parity chunks is updated. To update a parity chunk, there are two choices, *read-modify-write* (RMW) and *read-construct-write* (RCW). For performance consideration, traditional parity update algorithm works as follows. If a parity chunk is to be updated, the system first determines all the data chunks that are encoded into this parity chunk, and then checks which data chunks are already in memory and which need be read from disks (namely *pre-read*). After that, the system calculates the number of pre-reads required by RMW and RCW for updating the parity chunk, and then chooses the one with fewer pre-reads. RAID4S-modthresh [16] modifies the algorithm in RAID4S scheme by choosing RMW with a higher probability than RCW for small-writes, while still sticking to the traditional algorithm for large-writes.

We note that the existing parity update algorithm does not consider the dynamics of I/O queues on different disks in a RAID array, so it may make the distribution of the length of I/O queues on different disks be highly skewed, and thus degrades the system performance as the disk with the longest I/O queue may become the bottleneck. From our experiments, the skewness does exist in many I/O traces, and sometimes the performance degradation is also severe. This motivates us to explore a parity update algorithm by taking the state of I/O queues on a RAID into consideration, so as to lighten the skewness of I/O queues by balancing the workload on all disks while updating parity chunks. In a summary, we make the following contributions in this paper.

- We propose a balanced parity update algorithm, BPU, which further takes the state of I/O queues and online I/O workload into consideration when selecting RMW or RCW for parity update. In BPU, we set multiple thresholds to measure the number of pre-reads required for updating a parity chunk, the degree of the skewness of I/O queues, and the online I/O workload, and then select RCW or RMW according to these thresholds and real-time I/O workload so as to achieve a higher load balance and lower write overhead for parity update.
- We implement a prototype system by modifying the Linux kernel 4.0.2, and deploy RAID-5 with BPU on a server with nine SSDs. Our experiments show that the skewness among I/O queues does exist while updating parity chunks and it degrades system performance.
- We conduct extensive experiments on the prototype system with real-world traces. Our experiments show that BPU reduces the average response time by up to 10% compared to traditional parity update algorithm. We also provide multiple insights for regulating the parameters to optimize the performance of BPU.

The rest of this paper is organized as follows. In Section II, we present the background of RAID with erasure code and two parity update methods. In Section III, we motivate the design of BPU. We present the design details of BPU and the prototype implementation in Section IV and Section V, respectively. We evaluate the performance of BPU with real-world workloads in SSD RAID-5 system in Section VI. Finally, we conclude this paper in Section VII.

II. Background

A. RAIDs

A RAID consists of multiple physical disks to provide large storage capacity and high parallel I/O bandwidth. Disks in a RAID system are organized into many *stripes*, each of which consists of multiple logical units, named *chunks*. Chunks in a stripe are distributed across disks, with one on each. RAID provides high parallel I/O bandwidth because multiple I/O operations can be concurrently executed on different disks. Besides, RAID can also provide data redundancy to against disk failure. The commonly used approaches for data redundancy are mirroring data from one disk to another disk or encoding data chunks into *parity chunks*.

RAID-5 and RAID-6 are two commonly used RAIDs to against disk failures. RAID-5 tolerates one disk failure with XOR encoding, while RAID-6 tolerates two disk failures with different codes, such as RDP, P-code, EVEN-ODD and CRS code. In the following, we call the set consisting of a parity chunk and the data chunks that are used to encode this parity chunk the *parity chain* of this parity chunk. In practical implementation, the parity chunks are rotationally distributed among all disks for load balance consideration.

B. Read-modify-write and Read-construct-write

In RAID systems, the parity chunk needs to be updated synchronously, if any data chunk in its parity chain is updated, so as to ensure data consistency and data availability. There are two methods for parity updating, *read-modify-write* (RMW) and *read-construct-write* (RCW).

RMW works as follows. When some data chunks in a parity chain are updated, it first reads the old data chunks that are to be updated and the old parity chunks in the same parity chain from disks into memory. Because the newly updated data chunks are already in memory, they can be accessed directly from memory. Then, RMW updates the parity chunk according to Equation (1). Finally, the new parity chunks and new data chunks are written back into disks synchronously.

$$Parity_{new} = Data_{old} \oplus Parity_{old} \oplus Data_{new}. \qquad (1)$$

Unlike RMW, RCW uses the updated data chunks and other data chunks that are not updated but belong to the same parity chain with the updated data chunks to reconstruct a new parity chunk. Therefore, RCW just reads all non-updated data chunks in the parity chain from disks into memory, and then generates new parity chunks by XOR-summing these non-updated data chunks and newly updated data chunks with Equation (2). At last, new data chunks and new parity chunks are written from memory into disks.

$$Parity_{new} = Data_{non-updated} \oplus Data_{new}. \qquad (2)$$

Note that both RMW and RCW need extra pre-read requests to read data/parity chunks from disks into memory to prepare the updating of parity chunks. In the following of this paper, we shortly name the number of data/parity chunks that need to be read from disks into memory to update a parity chunk as *the number of pre-reads*. Pre-read introduces extra I/O workload on a RAID and degrades the system performance, especially for systems with many random small writes.

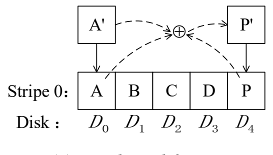
III. Motivation

In this section, we first introduce the traditional parity update algorithm for RAIDs with erasure codes, then analyze the skewness of I/O queues on all disks, and finally present the motivation of our new algorithm.

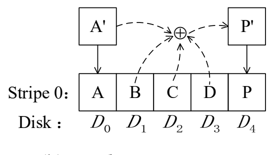
A. Traditional Parity Update Algorithm

As stated in Section II-B, there are two ways, RMW and RCW, to update parity chunks. For performance consideration, the traditional parity update algorithm firstly calculates the number of pre-reads required by RMW and RCW, and then selects the one with fewer pre-reads so as to reduce the I/Os.

Fig. 1 shows an example to illustrate traditional update selection algorithm. In Fig. 1, a stripe consists of four data chunks A, B, C, D and one parity chunk P, which are distributed on disks $D_0, D_1, ..., D_4$. Now suppose that there comes a write request to update data chunk A to A'. We also suppose that all data/parity chunks do not exist in memory. RMW, as shown in Fig. 1(a), needs to read two chunks A and P, and updates parity chunk P to P' with $P' = A \oplus A' \oplus P$. So



(a) *read-modify-write*



(b) *read-construct-write*

Fig. 1. Two parity update ways in RAID for updating chunk A.

RMW incurs two pre-reads, which access disks D_0 and D_4, respectively. For RCW, as depicted in Fig. 1(b), it needs to read three chunks B, C and D from disks into memory to update parity chunk P to P' with $P' = A' \oplus B \oplus C \oplus D$. So RCW incurs three pre-reads, which access disks D_1, D_2, and D_3, respectively. To minimize the total number of I/Os, traditional parity update algorithm chooses RMW in this example as it requires one fewer pre-reads when updating parity chunk P.

B. *Skewness of I/O Queues*

Note that the traditional parity update algorithm chooses either RMW or RCW by considering only the number of pre-reads required, so as to minimize the total number of I/Os while updating parity chucks. However, it neglects the skewness of I/O queues on different disks, and thus may choose the one (RMW or RCW) which introduces pre-reads to the disks with heavier I/O workloads. As a result, it aggravates the skewness of I/O queues, and make the I/O workloads on different disks more unbalanced.

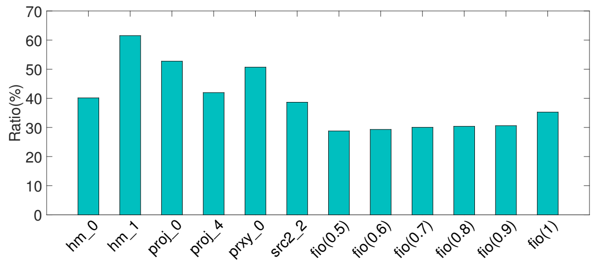


Fig. 2. The ratio of parity update requests that need to access the disk with the maximal length of I/O queue when using the traditional algorithm.

To further illustrate the skewness problem of I/O queues, we still take the RAID array shown in Fig. 1 as an example. Suppose that the I/O workloads on disk D_0 and D_4 are much heavier than those on disks D_1, D_2, D_3. So RMW will aggravate the skewness of I/O queues, and prolong the system response time. Even though RCW needs one more pre-reads than RMW, it accesses the disks with lighter I/Os, so the update of P can be finished earlier.

To validate the existence of skewness of I/O queues while updating parity chunks, we embed the traditional algorithm on a server deployed a 7+1 RAID-5. We run four real-world workloads on the system (see Table II in Section VI-A for workload details). To further examine the skewness of the I/O queues, we also conduct experiments with synthetic workloads, which write 10GB data to RAID-5 in 4KB unit issued by *fio*[1]. And the result is shown in Fig. 2. In Fig. 2, in terms of real-world workloads, we see that ratio reaches to 38.7%-52.7% for the four write intensive traces, and reaches to 41.9%-61.5% for the two read-intensive workloads. We also find that almost 10.2%-33.8% of all skewness are greater than 0.25ms, and almost 10% cases are even greater than 0.5*ms*. And for synthetic workloads, we find that ratio almost reaches to 28.8%-35.3% with varying write ratio. This implies that skewness does exist in real-world applications.

C. *Motivation*

Traditional parity update algorithm chooses either RMW or RCW with fewer pre-reads so as to reduce total I/Os, but it may aggravate the skewness of I/O workloads. On the other hand, we can also choose the one from RMW and RCW, which balances the I/O workloads on different disks, but this may lead to more pre-reads and increase the total workload to the RAID. So it is important to balance the trade-off between the number of pre-reads and the skewness of I/O queues. To analyze the tradeoff, we first characterize the lengths of I/O queues on different disks, and then study the impacts of I/O workloads and I/O queue skewness on system performance. Finally, our goal is to design a new parity update algorithm which selects RMW or RCW by considering both I/O workloads and I/O queue skewness, so as to reduce the overall response time of user requests and parity update.

IV. A Balanced Parity Update Algorithm

In this section, we propose a balanced parity update algorithm BPU, which takes the number of pre-reads, I/O skewness and real-time workloads into consideration. We first present some basic definitions related to pre-read, I/O skewness, and real-time workload on a RAID, then explain how to combine pre-read with skewness in BPU, and at last we present the design of BPU.

A. *I/O Queue Length*

There are two types of I/O requests, read and write, in the I/O queue of a disk. Because the access latencies of reading a chunk and writing a chunk are different, we formulate the queuing delay of an I/O queue as

$$L = N_R \times A_R + N_W \times A_W, \tag{3}$$

where N_R and N_W denote the number of read requests and the number of write requests in a queue, respectively, A_R and A_W denote the access latencies of processing a read request and a write request on a disk, respectively. Note that A_R and A_W depend on storage mediums, and it is very difficult to accurately predict the values of A_R and A_W. Note that Equation (3) formulates the latency of individual disk. So it is applicable to RAIDs with heterogeneous disks.

Suppose that there are n disks $D_0, D_1, ..., D_{n-1}$ in a RAID. We define its I/O queue length as a vector

$$L_{RAID} = (L_0, L_1, ..., L_{n-1}), \tag{4}$$

where L_i is the length of the I/O queue on disk D_i.

B. Pre-read for Updating Parity Chunk

If some data chunks in a parity chain are updated, the corresponding parity chunk should be updated to keep data consistency and data availability. We can use either RMW or RCW to update a parity chunk according to Equation (1) or (2). To complete the updating, we should first perform pre-read to read some data chunks and/or parity chunk from disks into memory. We define $N_{R_{rmw}}$ and $N_{R_{rcw}}$ as the numbers of pre-reads for updating a parity chunk with RMW and RCW, respectively. After computing $N_{R_{rmw}}$ and $N_{R_{rcw}}$, we can easily derive the difference of the numbers of pre-reads induced by RMW and RCW, and we have $\Delta_{pre} = |N_{R_{rmw}} - N_{R_{rcw}}|$.

C. Tradeoff between Pre-read and Skewness

The basic idea of BPU is that we should pay more attention to the number of pre-reads when the workload on a RAID is heavy or the I/Os on all disks are evenly distributed or only lightly skewed; Otherwise, we should focus on the skewness of I/O queues.

We use the average length of I/O queues on all disks to dynamically measure the online workload on a RAID, i.e., $L_{avg} = \frac{\sum_{i=0}^{n-1} L_i}{n}$. To capture the workload status, we define two thresholds, T_{heavy} and T_{light}, and classify workloads into three states, *Overloaded, Normal and Lightloaded* according to the following formula.

$$state = \begin{cases} Overloaded, & L_{avg} \geq T_{heavy}; \\ Normal, & T_{heavy} > L_{avg} \geq T_{light}; \\ Lightloaded, & T_{light} > L_{avg}. \end{cases} \quad (5)$$

To measure the skewness of I/O queues while updating parity chunk, we define $L_{max}^{rmw} = max\{L_i|$ *Disk* D_i *is accessed to update a parity chunk with RMW* $\}$, $L_{max}^{rcw} = max\{L_i|$ *Disk* D_i *is accessed to update a parity chunk with RCW* $\}$. With L_{max}^{rmw} and L_{max}^{rcw}, we define $\Delta_l = |L_{max}^{rmw} - L_{max}^{rcw}|$.

L_{max}^{rmw} is the maximum I/O queue length on the disks which will be accessed when we use RMW to update a parity chunk. Similarly, L_{max}^{rcw} is the maximum queue length when we use RCW to update a parity chunk. So Δ_l is the difference of skewness when we update a parity chunk with RMW and RCW, respectively. Intuitively, to update a parity chunk, we should pay more attention to the number of pre-reads when Δ_l is small; Otherwise, we should pay more attention to skewness. So we define a threshold T_{skew} to identify whether Δ_l is small or large. We also define a threshold T_{pre} to identify whether Δ_{pre} is small or large. If Δ_{pre} is large, we should select RMW or RCW which incurs fewer pre-reads; Otherwise, we should pay more attention to the skewness of I/O queues.

D. BPU: A Balanced Parity Update Algorithm

Now we are ready to present BPU for updating a parity chunk. It bases on the following rules:

1) When the workload is *Overloaded*, we should reduce the coming I/O loads rather than relieve skewness with more I/O loads.
2) When the workload on a RAID is *Normal*, we should take both of Δ_{pre} and Δ_l into consideration to make a balance between I/O overhead and I/O skewness.
   - When $\Delta_l \geq T_{skew}$, which means that the difference of skewness with RMW and RCW is large, we should pay more attention to the skewness; otherwise, we could ignore the skewness to focus on I/O overhead, i.e, select one from RMW and RCW with fewer pre-reads.
   - When $\Delta_{pre} > T_{pre}$, which means that the one from RMW and RCW with more pre-reads will add considerable more load to a RAID. So we should reduce pre-reads instead of relieving I/O skewness.
3) When the workload is *Lightloaded*, we could focus on relieving I/O skewness even though with more I/O loads.

E. Time Complexity of BPU

Suppose that a RAID consists of n disks and is deployed with an (n, r)-erasure code, i.e. there are r data chunks and $n - r$ parity chunks in a stripe. So the RAID tolerates $n - r$ disk failures. For practical settings, $r > n - r$, and usually $r \geq 2(n - r)$. The most common used ones are RAID-5 and RAID-6 with $n - r = 1$, or 2 respectively.

The traditional parity update algorithm checks the encoding equation of a parity chunk, then counts $N_{R_{rcw}}$ and $N_{R_{rmw}}$, and selects the smaller one. So it needs to go through the encoding equation and checks which data/parity chunks are in memory. It checks n chunks in memory. Apart from $N_{R_{rcw}}$ and $N_{R_{rmw}}$, BPU needs extra computation of L_{max}^{rcw} and L_{max}^{rmw}, i.e., n queue lengths. Because all of $N_{R_{rcw}}$, $N_{R_{rmw}}$, L_{max}^{rcw} and L_{max}^{rmw} are counted in memory, the difference of time complexities between BPU and traditional update algorithm is negligible compared to the time to complete I/O requests.

F. Extra I/O Loads of BPU

The extra I/O load for updating a parity chunk With BPU is reading 0 or Δ_{pre} data/parity chunks, which is limited by the threshold of T_{pre}. But in real systems, it also strongly depends on application workloads. So it is hard to give a theoretical analysis of extra I/O loads brought by BPU. For the applications with light I/O loads but heavy skewness, we should select large T_{pre}, but conversely for ones with heavy I/O loads but light skewness, we should select small T_{pre}.

G. Summary

The basic idea of BPU is straightforward. But its performance strongly depends on the selections of T_{pre}, T_{skew}, T_{heavy}, T_{light} and real time workload on a RAID. In Section VI, we will present some insights for the selections of these parameters related to workloads with experiments.

V. System Design

To evaluate the performance of BPU, we modified Linux kernel 4.0.2 and implemented a prototype system of RAID-5 with BPU on a server with 9 SSDs. We implemented traditional parity update algorithm and BPU on the prototype system respectively and run some common used traces on it to compare their performances. In this section, we first present an overview of RAID architecture and a diagram of its I/O flow, and then the detail implementation of BPU.

A. An Overview of RAID Architecture

A RAID typically consists of three components, *RAID controller*, *I/O queue on each disk* and *physical disk array*, as shown in right part of Fig. 3. RAID controller receives read or write (abbr. R/W) requests from workloads, interprets them according to the data layout in disk array and transfers them to I/O queues. An I/O queue stores the requests which are waiting for being scheduled by the I/O driver of each disk. Disk array is a group of storage medium, such as HDD or SSD. The HDDs or SSDs in a disk array may be heterogeneous.

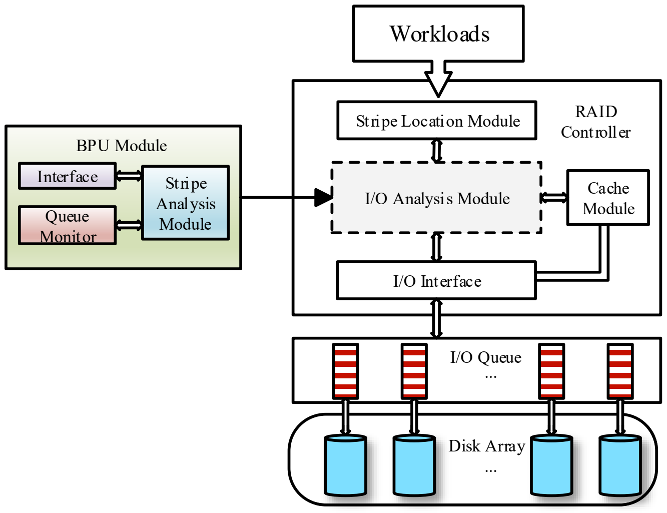


Fig. 3. The Overview of RAID Architecture with BPU.

B. RAID Controller

A RAID Controller typically consists of four components, *Stripe Location Module*, *I/O Analysis Module*, *Cache Module* and *I/O Interface*. Its workflow is mainly as follows.

Stripe Location Module analyzes the received requests to get their target data chunks and stripes according to the data layout in disk array, and then sends the stripes along with target chunks to I/O *Analysis Module*.

I/O Analysis Module traverses a stripe and decides which data/parity chunks should be read from or write into disk array. If a parity chunk is to be updated, *I/O Analysis Module* selects one from RMW and RCW with an algorithm. At last, it sends R/W requests to I/O Interface.

Cache Module temporarily stores the data chunks accessed by R/W requests and the parity chunks to be updated to keep data consistency and data availability.

I/O Interface receives from *I/O Analysis Module* that which data/parity chunks should be read from or write into a RAID. It then adds the requests into the corresponding I/O queues of disks, where the requests are waiting for being executed.

C. I/O Analysis Module with BPU

In our prototype system, we embed BPU in *I/O Analysis Module* as an algorithm to select one from RMW and RCW to update parity chunks. Apart from *Stripe Analysis Module* in traditional *I/O Analysis Module*, BPU adds to it two components, *Queue Monitor* and *Interface*. *Interface* is in charge of setting some parameters, such as the thresholds in BPU algorithm and the access latency of disks. To increase flexibility of the module, our system supports modifying these parameters dynamically through *Interface*. *Queue Monitor* captures real-time information of I/O Queues, including the numbers of read/write requests on all disks. The information in *Queue Monitor* is consistent with the I/O queue all the time.

Stripe Analysis Module first traverses the stripe to find out related information of each chunk to be accessed. For a request to read a data chunk, the data chunk will be fetched directly from memory if it resides in *Cache Module*; otherwise, *Stripe Analysis Module* will send the read request to *I/O Interface*. To write a data chunk, the data chunk and the parity chunk in the same stripe should be updated simultaneously. So Stripe Analysis Module will look up *Cache Module* to see which chunks in the parity chain residing in memory and select RMW or RCW to update the parity chunks. Once RMW or RCW is selected, *Stripe Analysis Module* sends the requests to read or write data/parity chunks to *I/O Interface*. We embed BPU algorithm into *Stripe Analysis Module* to support balanced parity update algorithm.

D. Implementations

We have implemented a RAID-5 with BPU based on the *software-raid* [2] within Linux OS. To realize BPU, we make the following modifications on the source code in *md* layer of Linux kernel 4.0.2.

- We add two arrays, *Read_IO* and *Write_IO*, to record the number of chunks to be read and written on each disk of a RAID. Each array contains N elements, where N is the number of disks in a RAID. Each element occupies 4 Bytes. We support atomic operation for these two arrays to ensure data consistency.
- We also add four variables to record four thresholds T_{heavy}, T_{light}, T_{skew} and T_{pre} in BPU algorithm. Each of them occupies 4 Bytes. We can modify them dynamically by *Interface* in BPU Module.
- We also add two arrays *Read_latency* and *Write_latency* to record read latencies and write latencies of all disks of a RAID. So we can compute the lengths of I/O queues on all disks of a heterogeneous RAID. Each element of two arrays occupies 4 Bytes, and it was initialized when a RAID was configured.

VI. Performance Evaluation

In this section, we conduct extensive experiments with our prototype system via real-world workloads to compare the performances of BPU and traditional parity update algorithm. We also discuss about the parameter settings of T_{light}, T_{heavy}, T_{pre}, T_{skew} so as to optimize BPU's performance. We first introduce the system configuration and the workload traces used in our experiments. Next, we show the performance improvement of BPU with different parameters, compared with traditional parity update algorithm. At last, we provide several insights to set the thresholds used in BPU. By default, we averaged over 3 runs for all our experiments unless we state otherwise.

A. System Configuration and Workloads

We conduct our experiments on a DELL^{TM} PowerEdge T620 server with four 2.50GHz Intel Xeon(R) E5-2609 CPUs and 8GB memory. The operating system is Ubuntu 15.04, 64-bit, which is installed in a single SSD. We configure eight SSDs as a 7+1 SSD RAID-5 by *mdadm*[10] command. Table I shows the detailed specification of the SSDs.

TABLE I
SPECIFICATION OF SSDS

Specification	SSD
Manufacture	Intel
Model	DC S3500 Series
Flash Memory	MLC
Capacity	120GB
Form Factor	2.5-inch
Interface	SATA 3.0 6Gb/s
SEQ. Read	445MB/S
SEQ. Write	135MB/s

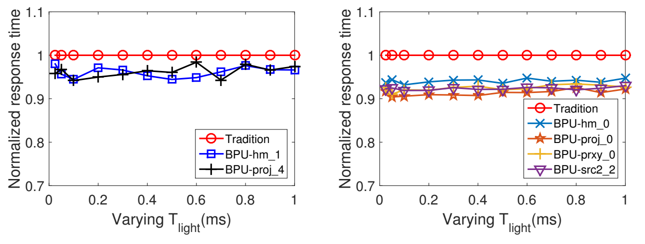
We consider several real-world workloads[9] collected from enterprise data center. Table II shows the detailed statistics of these traces. These block-level traces are named as $< server\_volume >$, which means that it is collected from which server and which data volume, e.g. prxy_0 workload is generated from volume 0 in Web proxy server. The traced period was one week originally. We compress the period to half one hour for intensive performance evaluation.

We design BPU to improve the performance of parity update in RAIDs. So we select four traces(e.g. hm_0, proj_0, prxy_0 and src2_2) because they are write-intensive. We can find from Table II that the minimal ratio of write request is 0.65, and the maximal ratio reaches up to 0.97. Furthermore, we also select two traces, hm_1 and proj_4, to evaluate its performance in read-intensive environment.

B. Impacts of Parameters

In this subsection, we mainly discuss the impacts of different parameters to the BPU. According to the various parameters configuration, we evaluate the performance of the traditional parity update algorithm and BPU. Because there are too many combinations for setting the parameters of T_{light}, T_{heavy}, T_{pre} and T_{skew}, we first execute the six traces with traditional parity update algorithm to get some basic statistics about the number of pre-reads, average I/O queue length, maximal I/O queue length. From these statistics, we can get some reasonable settings of them in our experiments.

$\boldsymbol{T_{light}}$**:** We evaluate the performance of BPU with different settings of T_{light}. We fix $T_{pre} = 3$, $T_{heavy} = 4ms$ for every heavy workload and $T_{skew} = 50us$ for light skewness of I/O queues. The settings of T_{heavy} and T_{skew} in this subsection



(a) *read-intensive workloads* (b) *write-intensive workloads*

Fig. 4. The normalized average response time under varying T_{light}.

are to make BPU have more chances to select RCW and RMW with light I/O skewness. We set T_{light}=0.025, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 to $1ms$. Fig. 4 shows the normalized average response time of requests when using traditional parity update algorithm and BPU under varying T_{light}. Because the response time of traditional algorithm does not depend on T_{light}, we always set the normalized average response time of which is 1.

As shown in Fig. 4(a), compared with traditional algorithm, BPU reduces the average response time by 6% at most for read-intensive workloads. With the increase of T_{light}, the response time becomes unstable. However, the range of variation is still under 4%. Compared with performance for the read-intensive workloads, BPU performs better for the write-intensive applications. From Fig. 4(b), BPU improves the performance by 6%-10% compared to traditional algorithm. With varying T_{light}, the tendency of response time of BPU becomes flat for all write-intensive traces. Fig. 4(b) also reveals that BPU performs better for the workloads that contain heavier write requests.

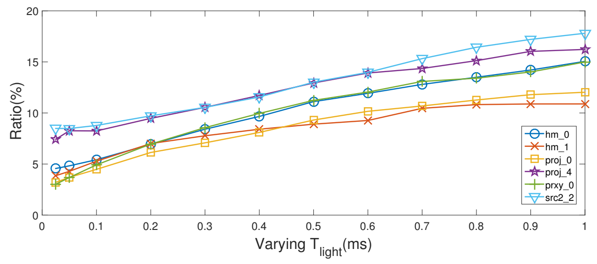


Fig. 5. The ratio of extra pre-reads.

When $L_{avg} < T_{light}$, we consider the workload on a RAID as *Lightloaded*. In this case, BPU always selects RCW or RMW with lighter I/O skewness regardless the number of pre-reads required for parity update, which will certainly increase the total I/Os on a RAID. But when the workload is *Normal*, we will limit the increase of the number of pre-reads by T_{pre} while choosing the one with lighter skewness, which makes the extra workload on a RAID be limited. As T_{light} increases, the system is considered to be *Lightloaded* more often, which makes the extra workload induced by pre-read be heavier. Fig. 5 shows the ratio of extra pre-reads with different settings of T_{light} when we choose RMW or RCW with lighter I/O skewness. The extra pre-reads ratio increases slowly when T_{light} increases from $25us$ to $1ms$. There are about a 10%

TABLE II
STATISTIC OF REAL-WORLD WORKLOADS

Workloads	Function	Total request numbers	Write ratio	Total I/O size	Average I/O size	The largest request size
hm_0	H/w monitor	3993316	0.65	30.44GB	8KB	0.72MB
hm_1	H/w monitor	609311	0.05	8.81GB	15KB	0.5MB
proj_0	Project dirs	4224524	0.88	153.24GB	38KB	0.7MB
proj_4	Project dirs	6465639	0.02	144.64GB	24KB	0.06MB
prxy_0	Web proxy	12518968	0.97	56.84GB	4KB	1MB
src2_2	Source control	1156885	0.69	62.07GB	56KB	0.13MB

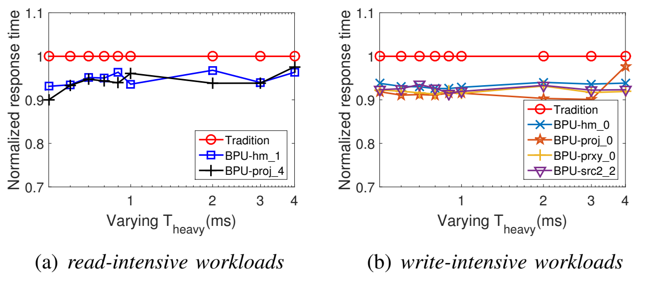


Fig. 6. The normalized average response time under varying T_{heavy}.

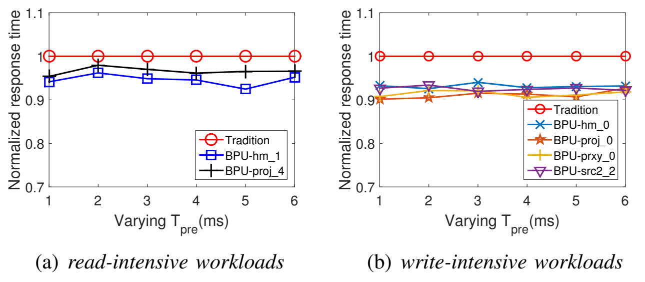


Fig. 8. The normalized average response time under varying T_{pre}.

ratio increase for all traces. But there are almost more 40% ratio that BPU selects RMW or RCW with lighter skewness. The gain of relieving skewness can well remedy the impact of extra pre-reads overhead. Hence, this is why BPU outperforms traditional algorithm a little as T_{light} increases.

From Fig. 4, we find that BPU almost performs best by setting $T_{light} = 0.1ms$. So in the following experiments, we always set $T_{light} = 0.1ms$ and evaluate the performance of BPU with different settings of other parameters.

***T_{heavy}*:** Now we present the performance of BPU with different settings of T_{heavy}. We fix T_{light}=100*us*, T_{skew}=50*us* and T_{pre}=3 as the same in last subsection. We vary T_{heavy} from 0.5, 0.6,,0.7, 0.8, 0.9, 1, 2, 3, 4 to 5 *ms*. Fig. 6 shows the performance comparison.

From Fig. 6(a), we find that the performance of BPU improves 10% than traditional algorithm for read-intensive workloads at most. However, the response time performs chaotic with different settings of T_{heavy}. As shown in Fig. 6(b), for write-intensive workloads, BPU improves the performance by 6%-10% compared to traditional algorithm. For example, compared the performance for hm_0 and proj_0, we find that BPU improves almost 8.2%-10% for proj_0 and 6%-7.3% for hm_0. Hence, we believe that BPU performs better for heavier write-intensive workloads. For BPU algorithm, the

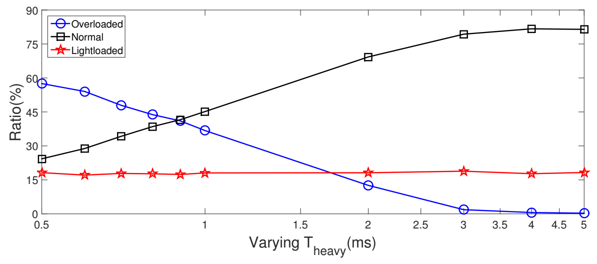


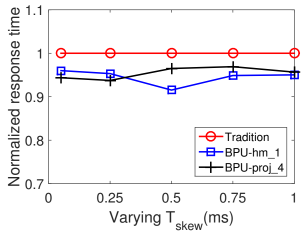
Fig. 7. The ratio of the state on RAID.

larger T_{heavy} is, the workload is considered to be *Normal* with higher chance. In terms of hm_0 workload, Fig.7 shows that the ratio of the RAID being *Overloaded* decreases from 57.5% to 1.8% when T_{heavy} increases from 0.5ms to 3ms. After that, the ratio keeps almost unchanged. With the increase of T_{heavy} from 0.5ms to 5ms, the ratio of a RAID being *Normal* increases. Due to the period variations, we find that about 21.5% of the parity updates are performed by one of RMW or RCW with lighter skewness when $T_{heavy} = 0.5ms$. And the ratio reaches 37.8% when $T_{heavy} = 5ms$. But BPU may generate more pre-reads when it selects the one of RMW or RCW with lighter skewness. According to the semantic analysis, however, the ratio of extra pre-reads increases a little(almost 1%-3%) when T_{heavy} increases. This is because other thresholds restrict the overhead increment of extra pre-reads even with higher probability that BPU selects the one of RMW or RCW with lighter skewness.

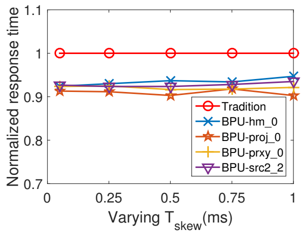
***T_{pre}*:** In this subsection, we study the impact of T_{pre} on the performance of BPU. We fix T_{light} = 100*us*, and T_{skew}=50*us*. We set T_{heavy} = 500*us* for read-intensive workloads and T_{heavy} = 1*ms* for write-intensive applications respectively. We vary T_{pre} from 1, 2, 3, 4, 5 to 6. Fig. 8 shows the average response time to updating parity chunk.

As shown in Fig. 8(a), BPU improves the performance by 7.6% at most for read-intensive workloads. When T_{pre} increases form 1 to 6, the response time presents a gently fluctuation. From Fig. 8(b), BPU shows a better improvement for write-intensive applications than read-intensive traces, which shortens the response time by 9.9% at most. We take proj_0 as an example, the average response time of traditional algorithm is 13.8ms. With T_{pre} increases, the response time is about 12.6ms and shows a mild tendency. We observe similar trend for other workloads.

With T_{pre} increases, BPU has more chance to choose RMW or RCW with lighter skewness. In the case of proj_0 workload, there are more 20% ratio of the parity updates are performed by one of RMW or RCW with lighter skewness.



(a) *read-intensive workloads*



(b) *write-intensive workloads*

Fig. 9. The normalized average response time under varying T_{skew}.

But BPU generates a little(almost 2.5%-6.4%) extra pre-reads than traditional algorithm. Hence, the extra overheads has a slight impact on the performance. And BPU improves the performance than traditional algorithm within a gently fluctuation as T_{pre} increases.

$\boldsymbol{T_{skew}}$**:** We now evaluate the impact of T_{skew} on the performance of BPU. We fix $T_{light} = 100us$, and $T_{pre} = 3$. We set $T_{heavy} = 500us$ for read-intensive workloads and $T_{heavy} = 1000us$ for write-intensive applications respectively. We vary T_{skew} from 0.05, 0.25, 0.5, 0.75 to $1ms$. The performance comparison is shown in Fig. 9.

BPU consistently outperforms traditional algorithm under all settings of T_{skew}. The improvement reaches 5.4%-7.6%, 4%-8.5%, 8.3%-9.8%, 3.1%-6.3%, 7.7%-8.5% and 6.5%-7.7% for hm_0, hm_1, proj_0, proj_4, prxy_0 and src2_2, respectively. When T_{skew} increases from 0.05 to $1ms$, the response time presents a flat outlook for all traces. When T_{skew} increases, BPU selects the one from RCW or RMW with lighter skewness only when skewness becomes heavier. As a result, the gain of response time reduction comes from the balancing of I/O queues can well compensate the loss of performance caused by more pre-reads. We believe that this is the reason why BPU performs a little better as T_{skew} increases.

C. Summary

Based on our experiments, we show that BPU outperforms traditional parity update algorithm. With more refinements of thresholds, BPU shows a stable performance improvement. Based on our experiments, we learn the following insights to regulate the settings of $T_{light}, T_{heavy}, T_{pre}$ and T_{skew}.

1) T_{light} should be set to be small and T_{heavy} should be set to be large. So BPU has a higher probability to balance the pre-reads and skewness.
2) T_{pre} usually is configured as the half of the disk number in a RAID. Hence, BPU has more chance to make the tradeoff of pre-reads and skewness.
3) T_{skew} should be set to be a little larger than the average skewness. So the benefit from lightning the skewness can well compensate the extra pre-reads induced by BPU.

VII. Conclusion

In this paper, we proposed a balanced parity update algorithm, BPU, which takes pre-reads, I/O skewness and real-time workload on a RAID into consideration. To validate the efficiency of BPU compared to traditional parity update algorithm, we modified Linux kernel 4.0.2 and embedded BPU into the kernel to realize a RAID-5 system with 7+1 SSDs. Our experiments showed that the skewness of I/O queues does exist when we update parity chunks by using the traditional parity update algorithm, which prolongs the response time to parity update. We also conducted extensive experiments to optimize the performance of BPU by adjusting some basic thresholds. Experimental results showed that BPU could reduce the average response time by up to 10% compared with traditional parity update algorithm.

Acknowledgements

This work was supported by National Nature Science Foundation of China under Grant No. 61379038 and No. 61303048 and Huawei Innovation Research Program under Grant No. HIRPO20140301.

References

[1] fio. http://freecode.com/projects/fio.
[2] Software RAID. https://en.wikipedia.org/wiki/RAID#Software-based.
[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference*, pages 57–70, 2008.
[4] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures. *IEEE Transactions on Computers,*, 44(2):192–202, 1995.
[5] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 181–192. ACM, 2009.
[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.
[7] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.
[8] J. L. Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *FAST*, volume 5, pages 16–16, 2005.
[9] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.
[10] J. Østergaard and E. Bueso. The Software-RAID HOWTO. *Acessado em 04/04/2003, disponível*, 2000.
[11] D. A. Patterson, G. Gibson, and R. H. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, volume 17. ACM, 1988.
[12] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *FAST*, volume 7, pages 17–23, 2007.
[13] J. S. Plank, M. Blaum, and J. L. Hafner. SD Codes: Erasure Codes Designed for How Storage Systems Really Fail. In *FAST*, pages 95–104, 2013.
[14] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications. In *Fifth IEEE International Symposium on Network Computing and Applications*, pages 173–180. IEEE, 2006.
[15] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you? In *FAST*, volume 7, pages 1–16, 2007.
[16] M. Sevilla, R. Wacha, and S. A. Brandt. RAID4S-modthresh: Modifying the Write Selection Algorithm to Classify Medium-Writes as Small-Writes. Technical report, Technical Report UCSC-SOE-12-10, University of California, Santa Cruz, 2012.
[17] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Peer-to-Peer Systems*, pages 328–337. Springer, 2002.
[18] L. Xu and J. Bruck. X-Code: MDS Array Codes with Optimal Encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.