PAPER *Special Issue on Dependable Computing*

A Self-Adjusting Destage Algorithm with High-Low Water Mark in Cached RAID5

Young Jin NAM[†] ***and*** **Chanik PARK**[†]**,** ***Nonmembers***

SUMMARY The High-Low Water Mark destage (HLWM) algorithm is widely used to enable cached RAID5 to flush dirty data from its write cache to disks due to the simplicity of its operations. It starts and stops a destaging process based on the two thresholds that are configured at the initialization time with the best knowledge of its underlying storage performance capability and its workload pattern which includes traffic intensity, access patterns, etc. However, each time the current workload varies from the original, the thresholds need to be re-configured with the changed workload. This paper proposes an efficient destage algorithm which automatically re-configures its initial thresholds according to the changed traffic intensity and access patterns, called adaptive thresholding. The core of adaptive thresholding is to define the two thresholds as the multiplication of the referenced increasing and decreasing rates of the write cache occupancy level and the time required to fill and empty the write cache. We implement the proposed algorithm upon an actual RAID system and then verify the ability of the auto-reconfiguration with synthetic workloads having a different level of traffic intensity and access patterns. Performance evaluations under well-known traced workloads reveal that the proposed algorithm reduces disk IO traffic by about 12% with a 6% increase in the overwrite ratio compared with the HLWM algorithm.

key words: *delayed write, high-low water mark, destage algorithm, write cache, RAID5*

1. Introduction

Despite its prevalence, RAID5 [1] has suffered from a small write problem, which refers to the phenomenon that RAID5 requires four disk accesses to write a single disk block: old parity/data reads and new parity/data writes. To overcome the small write problem in RAID5, several solutions have been proposed [2]–[6]. One utilizes the non-volatile write cache [2]. Since the service of a write request entails writing data onto the write cache, it can be performed quickly. The written (*i.e.*, dirty) data in the cache should be flushed into physical disks when the write cache occupancy level arrives at a pre-defined threshold. The operation of flushing dirty data is called a destaging process, which is usually initiated by a destage scheduler or a destage algorithm.

When designing a destage algorithm, we generally account for two aspects: how to elaborately determine its thresholds to manage the destaging process (e.g., starting or stopping destaging), and how to decide which dirty cache entries are to be destaged. This paper mainly focuses on the first aspect.

Two destage algorithms have been available in terms of manipulating destaging thresholds: the High-Low Water Mark (HLWM) algorithm [2] and the Linear Threshold (LT) algorithm [6]. The HLWM algorithm is based on two destaging thresholds: the High Water Mark (HWM) and the Low Water Mark (LWM). It starts the destaging process when the current write cache occupancy level goes up to HWM. Conversely, it stops destaging when the write cache occupancy level goes down to LWM. The LT algorithm defines more than two thresholds. As the current write cache occupancy level reaches each higher threshold, it accelerates the destaging process by including more dirty entries in a single destaging write operation. Even though this algorithm outperforms the HLWM algorithm, it requires a complicated operation to compute the destaging costs for all the dirty entries at every disk head move. Of these, the HLWM algorithm is widely used to enable a cached RAID5 to flush dirty data from its write cache to disks due to the simplicity in its operations. However, the maintenance of the desirable threshold values of the HLWM algorithm is time-consuming. That is, each time the current workload varies from the original workload, the thresholds need to be re-configured with the changed workload.

This paper proposes an efficient destage algorithm which adaptively reconfigures its thresholds according to the changed workload, called adaptive thresholding. The core of adaptive thresholding is to define its thresholds as the multiplication of the referenced increasing and decreasing rates of the write cache occupancy level and the time required to fill and empty the write cache. After implementing it upon an actual RAID system, we verify the ability of the auto-reconfiguration of the proposed algorithm by using synthetic workloads and performance enhancements by using well-known traced workloads.

The remainder of this paper is organized as follows. Section 2 provides the design of the proposed algorithm. Section 3 describes the implementation details of the proposed algorithm upon an actual RAID system. Section 4 compares its performance with that of the HLWM algorithm. Finally, we conclude this paper with Sect. 5.

Manuscript received April 7, 2003.
Manuscript revised July 7, 2003.

[†]The authors are with the Department of Computer Science and Engineering, Pohang University of Science and Technology, Pohang, 790–784 South Korea.

2. Design of a Self-Adjusting Destage Algorithm with High-Low Water Mark

This section addresses the major problem of the HLWM algorithm and then describes the proposed algorithm which automatically re-configures its thresholds. We begin by providing some nomenclature that will be used in the subsequent section.

2.1 Nomenclature

A RAID system has separate read and write caches, denoted by C_R and C_W. Next, C_R and C_W have N_R and N_W cache entries of the same size. A write IO issued by the host stores its data into non-dirty cache entries of C_W, and subsequently the cache entries become dirty. Denote with $N_W^D(t)$ the number of dirty cache entries within the write cache at time t, also called the current write cache occupancy level.

A read cache hit refers to a data reference that can be satisfied from either C_R or C_W. A write cache hit refers to a successful data store into available non-dirty cache entries within C_W. A write cache overwrite, shortly overwrite, refers to a data store into the current dirty cache entries, implying that it requires no additional non-dirty cache entries. A write cache miss refers to an unsuccessful data store into the write cache due to the lack of sufficient non-dirty cache entries.

Denote with $i(t)$ an increasing rate of the dirty cache entries at time t. $i(t)$ is a function of the current amount of write IOs and the overwrite ratio. Denote with $d(t)$ a decreasing rate of the dirty cache entries by the destaging process at time t. $d(t)$ is measurable only when the destaging process is active, while $i(t)$ is always measurable. Next, define $\delta(t)$, an effective decreasing rate of the number of dirty cache entries at time t, as $d(t) - i(t)$. We assume that a decreasing rate means $\delta(t)$, not $d(t)$, in the remainder of this paper. In sum, $i(t) = dN_W^D(t)/dt$ when the destaging process is inactive, and $\delta(t) = -dN_W^D(t)/dt$ when the destaging process is active.

2.2 Drawback of the HLWM Algorithm

The HLWM algorithm defines the two thresholds based on the write cache occupancy levels, which are High Water Mark (HWM) and Low Water Mark (LWM) denoted by $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$, respectively. $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ are rewritten as $\rho_H N_W$ and $\rho_L N_W$, where N_W is the number of the cache entries in the write cache and $0 \leqq \rho_L < \rho_H \leqq 1$.

Typically, $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ of the HLWM algorithm are configured by a system administrator at the initialization time with the best knowledge of its underlying storage performance capability and its workload pattern which includes traffic intensity, access patterns, etc. However, the initial workload pattern is subject to change due to numerous reasons, such as installations of new IO-intensive applications, the increase of storage usage, file system fragmentation, etc. That is, each time the current workload varies from the original workload, the thresholds need to be re-configured with the changed workload. In order to remove this management overhead, we need to devise a scheme to automatically reconfigure the threshold values according to any change with respect to the initial workload.

2.3 The Proposed Algorithm

To begin, we assume that $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ are given initially.

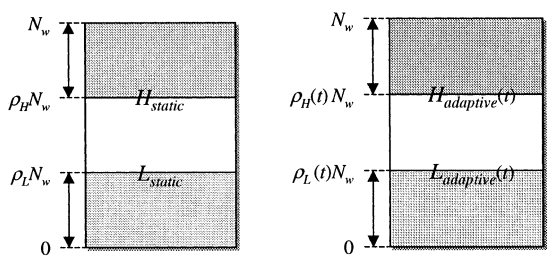
2.3.1 Adaptive Thresholding

Figure 1 compares the HLWM algorithm and the proposed algorithm in terms of how to maintain their thresholds. While the HWM and LWM are time-invariant in the HLWM algorithm, they are time-variant in the proposed algorithm, called adaptive thresholding. In order to design the adaptive thresholding, we first define the HWM and LWM of the proposed algorithm as follows:

$$\mathcal{H}_{adaptive}(t) = \rho_H(t) N_W, \tag{1}$$

$$\mathcal{L}_{adaptive}(t) = \rho_L(t) N_W, \tag{2}$$

where $\rho_H(t)$ and $\rho_L(t)$ are time-variant, and $0 \leqq \rho_L(t) < \rho_H(t) \leqq 1$ at any time t. We will derive $\rho_H(t)$ and $\rho_L(t)$. Denote with i_B and δ_B the referenced value of $i(t)$ and the referenced value of $\delta(t)$, respectively. The values of i_B and δ_B is automatically extracted from the initial workload by running the HLWM algorithm with $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ (See Sect. 2.3.2 for more details). Next, define T_H and T_L as the times required to fill the write cache from $\mathcal{H}_{static}$ with the referenced increasing rate i_B and to empty the write cache from $\mathcal{L}_{static}$ with the referenced decreasing rate δ_B, respectively. Thus, $T_H = \frac{(1-\rho_H)}{i_B} N_W$ and $T_L = \frac{\rho_L}{\delta_B} N_W$. The proposed algorithm starts a destaging process when the remaining time to fill up the write cache



(a) HLWM (b) Proposed

Fig. 1 Two thresholds of the HLWM algorithm and the proposed algorithm.

becomes T_H, which stops when the remaining time to empty the write cache becomes T_L. Therefore, given $i(t)$ and $\delta(t)$, the following equations should be valid: $i(t)T_H + \rho_H(t)N_W = N_W$ and $\rho_L(t)N_W - T_L\delta(t) = 0$. Finally, the time-variant $\rho_H(t)$ and $\rho_L(t)$ are written as $\rho_H(t) = 1-(1-\rho_H)\frac{i(t)}{i_B}$ and $\rho_L(t) = \rho_L\frac{\delta(t)}{\delta_B}$. In order to meet the inequality that $0 \leqq \rho_L(t) < \rho_H(t) \leqq 1$, $\rho_H(t)$ and $\rho_L(t)$ will be:

$$\rho_H(t) = max\{0, 1-(1-\rho_H)\frac{i(t)}{i_B}\}, \quad (3)$$

$$\rho_L(t) = min\{1, max\{0, \rho_L\frac{\delta(t)}{\delta_B}\}, \rho_H(t)\}. \quad (4)$$

Based on Eq. (1)–(4), the proposed algorithm adjusts $\mathcal{H}_{adaptive}(t)$ and $\mathcal{L}_{adaptive}(t)$ under the following four different types of changed workload patterns:

- **Case 1 -** $i(t) < i_B$ (when the increasing rate of the write cache occupancy level becomes slower than its referenced rate due to a reduced arrival rate of write requests from a host): $\mathcal{H}_{adaptive}(t) > \mathcal{H}_{static}$ and $\mathcal{L}_{adaptive}(t) > \mathcal{L}_{static}$
- **Case 2 -** $i(t) > i_B$ (when the increasing rate of the write cache occupancy level becomes faster than its referenced rate due to an increased arrival rate of write requests from a host): $\mathcal{H}_{adaptive}(t) < \mathcal{H}_{static}$ and $\mathcal{L}_{adaptive}(t) < \mathcal{L}_{static}$
- **Case 3 -** $\delta(t) > \delta_B$ (when the decreasing rate of the write cache occupancy level becomes faster than its referenced rate due to a better access pattern in terms of sequentiality or a spatial locality): $\mathcal{L}_{adaptive}(t) > \mathcal{L}_{static}$
- **Case 4 -** $\delta(t) < \delta_B$ (when the decreasing rate of the write cache occupancy level becomes slower than its referenced rate due to a worse access pattern in terms of sequentiality or a spatial locality): $\mathcal{L}_{adaptive}(t) < \mathcal{L}_{static}$

The overwrite ratio and disk IO traffic of the proposed algorithm are expected to be improved compared with the HLWM algorithm under the conditions of Case 1 and 3 by increasing the HWM and LWM values. In the transit from a given HWM (or LWM) to the lower HWM (or LWM), for example, the transit from Case 1 or Case 3 to its initial workload pattern, the proposed algorithm inevitably produces a larger amount of disk IO traffic than the HLWM algorithm.

The chances of a write cache miss of the proposed algorithm are expected to be lower than the HLWM algorithm under the conditions of Case 2 and 4 by decreasing the HWM and LWM values to reserve more space to absorb greater write IO traffic. Note that guaranteeing both a good write cache miss ratio and a high overwrite ratio together remains for the next step of our current work. To address this issue, we first need to devise a scheme to find the optimal values for $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$[†] initially. Next, we should carefully adjust the $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ not only to avoid any write cache miss, but also to maintain the write cache occupancy as high as possible.

The read cache hit ratios of the proposed algorithm and the LWM algorithm are expected to be the same; that is, the read cache hit ratio is little affected by the destage algorithm itself. The reason is that the read cache hit ratio at the write cache remains the same because the destage algorithm simply changes the states of the cache entries from dirty to clean, and the read cache hit ratio at the read cache remains almost the same because the destage algorithm allocates the least recently used cache entries for the old data and parity blocks.

In summary, the proposed algorithm performs better than the HLWM algorithm in terms of the overwrite ratio and the disk IO traffic under Case 1 and Case 3, but it behaves worse under Case 2 and Case 4. Next, the proposed algorithm performs better than the HLWM algorithm in terms of the write cache miss ratio under Case 2 and Case 4, but it works the same as the HLWM algorithm under Case 1 and Case 3. Note that the read cache hit ratios of both algorithm are equal.

2.3.2 Initialization of i_B and δ_B

Given $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$, Fig. 2 depicts how to extract the referenced values of $i(t)$ and $\delta(t)$ denoted with i_B and δ_B. Basically, i_B and δ_B are empirically obtained in the initialization time of the proposed algorithm. Before deciding these values, the proposed algorithm operates on the basis of the $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ like the HLWM algorithm. First, it logs a series of time-points each time the current write cache occupancy level reaches either $\mathcal{H}_{static}$ or $\mathcal{L}_{static}$. Denote with t_k^{HWM} and t_k^{LWM} the k-th time points when it reaches $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$, respectively. Assume that we have $\{t_1^{LWM}, t_1^{HWM}, t_2^{LWM}, t_2^{HWM}, t_3^{LWM}\}$, as shown in Fig. 2. Next, we compute the increasing slope i_k by dividing the value of $(\mathcal{H}_{static} - \mathcal{L}_{static})$ with the elapsed time from t_k^{LWM} to t_k^{HWM}. Similarly, we calculate the decreasing slope δ_k by dividing the value of

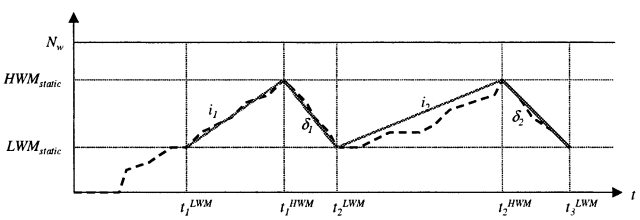


Fig. 2 Initializing the referenced values, i_B and δ_B.

[†]The optimal $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ refer to a pair of HWM and LWM thresholds which maintains the write cache occupancy level as high as possible without any write cache miss due to the lack of sufficient non-dirty cache entries.

($\mathcal{H}_{static} - \mathcal{L}_{static}$) with the elapsed time between t_k^{HWM} and t_{k+1}^{LWM}. Finally, i_B and δ_B are obtained by averaging the values of i_k and δ_k, respectively.

2.3.3 Updates of $\rho_H(t)$, $\rho_L(t)$, $i(t)$ and $\delta(t)$

According to the relationship given in Eq. (3) and (4), we mainly focus on the update of $i(t)$ and $\delta(t)$. The values of $i(t)$ and $\delta(t)$ are updated by a monitoring process every k seconds. Note that $\delta(t)$ is updated only when the destage is active. The values are computed by averaging three slopes of the increasing and decreasing rates of the write cache occupancy level that are obtained from the most recent three time-points associated with the HWM or LWM values, as well as the current time-point. These time-points can be denoted with $\{t_{last-1}^{HWM}, t_{last}^{LWM}, t_{last}^{HWM}, t_{cur}\}$ or $\{t_{last-1}^{LWM}, t_{last}^{HWM}, t_{last}^{LWM}, t_{cur}\}$. In addition, their associated write cache occupancy levels can be denoted with $\{H_{last-1}, L_{last}, H_{last}, C_1\}$ or $\{L_{last-1}, H_{last}, L_{last}, C_2\}$, respectively. Next, we compute three increasing and decreasing slopes with the above information and then average out the slopes. For the case of $\{t_{last-1}^{HWM}, t_{last}^{LWM}, t_{last}^{HWM}, t_{cur}\}$ and $\{H_{last-1}, L_{last}, H_{last}, C_1\}$, we have $i(t) = \frac{H_{last} - L_{last}}{t_{last}^{HWM} - t_{last}^{LWM}}$ and $\delta(t) = (\frac{H_{last-1} - L_{last}}{t_{last}^{LWM} - t_{last-1}^{HWM}} + \frac{H_{last} - C_1}{t_{cur} - t_{last}^{HWM}})/2$. Similarly, we can easily obtain $i(t)$ and $\delta(t)$ for the other case. For example, assume that we update $i(t)$ and $\delta(t)$ at $t = 30$, and we have four time-point of $\{10, 20, 25, 30\}$ and their associated write cache occupancy level of $\{70, 30, 70, 40\}$. Then, $i(t) = 40/5$ and $\delta(t) = (\frac{40}{10} + \frac{30}{5})/2$.

2.3.4 Other Issues

Designing a destage algorithm requires handling the maintenance of the delayed write list, the selection of dirty cache entries to be destaged, and the priority of the destaging process, etc. Each of these issues will be discussed in the following section.

3. Implementation

The proposed algorithm was implemented within an actual RAID system, called PosRAID [7]. Hardware components of the PosRAID encompass Pentium IV 1.3 GHz, 256 MB memory, two QLogic's QLA2200 Fibre Channel HBAs, 32 bit/33 MHz PCI bus, and eight 7200 rpm 9 GB Seagate Barracuda ST39175FC disks. However, PosRAID does not support a hardware XOR module. Software components of PosRAID include the following: VxWorks version 5.4 RTOS, a communication module which communicates with hosts and internal disks via Fibre Channel host bus adapters, a RAID engine module which maps a logical block address to a physical block address based on a given RAID architecture and manages the read and write caches, and a resource and/or configuration management module which allocates, deallocates, and configures all hardware and software resources within the RAID system.

3.1 Overall IO Operations

IO requests that arrive from a host system are first placed into the host IO request queue in an FIFO manner. Next, the host IO scheduler selects an IO request from the host IO request queue. In the case of the read IO, the host IO scheduler spawns a RAID read IO process. The RAID read IO process searches its data at the write cache and then the read cache. If valid data resides in the write or read cache, the cached data is immediately transferred to the host without accessing a physical disk. Otherwise, the data is retrieved from physical disks into the read cache and is then returned to hosts. In the case of the write IO, the host IO scheduler saves its data simply into the write cache and logs the write IO request into the delayed write list that will be referred by the destaging process. Next, it notifies its host of the completion of the current write request service. Unless sufficient write cache entries are available, called a write cache miss, the data transmission from the host will be postponed until enough space becomes available. The destaging process flushes the dirty cache entries into physical disks. The state transition of a write cache entry and the destaging process will later be described in detail. Each individual disk has a pair of the host request queue and the destage request queue. While disk IO requests made from a read request are placed into the host request queue of a disk, the destage read/write IO requests issued by the destaging process are lined up in the destage queue of a disk. Note that a host request queue is given a higher priority in service than a destage queue, *i.e.*, the requests in the destage request queue are processed only while the host request queue becomes empty. Each disk also contains its own IO scheduler. Even though a few efficient disk scheduling algorithms are currently available, we employ a simple C-LOOK disk scheduling algorithm combined with a batching scheme.

3.2 State Transition of a Write Cache Entry

Figure 3 depicts a state transition diagram of a write cache entry. Initially, the state of each entry is `NOT_USED` and `INVALID`. When the entry is allocated for storing dirty data from the host, its state is changed to `IN_USE` and `VALID/DIRTY`. If new dirty data arrives at the same location before destaging of the dirty data, it is overwritten to the entires, called the write cache overwrite. Conversely, if new dirty data arrives during destaging of the dirty data, it is stored separately by allocating different cache entries. At this point, its state is changed to `IN_USE` and `VALID/DDIRTY`. Subsequent dirty data at the same location will be overwritten, as previously. Once the destaging of a dirty

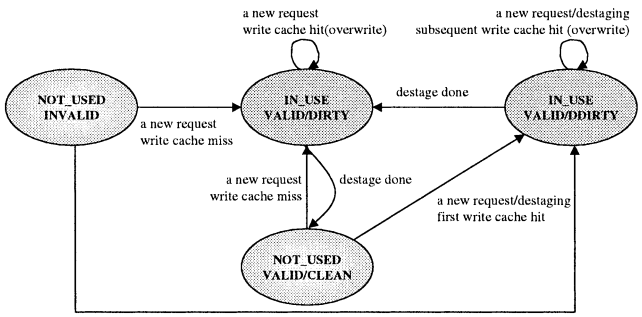


Fig. 3 State transition diagram of write cache entries.

data/entry completes, its state is changed to `NOT_USED` and `VALID/CLEAN`. In addition, if an associate cache entry of the `IN_USE` and `VALID/DDIRTY` state exists, then its state is changed to `IN_USE` and `VALID/DIRTY`.

3.3 Destage Process

The destage process flushes dirty write cache entries into physical disks. We have implemented the proposed algorithm and the HLWM algorithm as a key component of this destaging process. The destaging process starts when the current occupancy reaches its HWM and then stops when the current occupancy descends to its LWM. The destaging process first scans the list of delayed write IOs and then selects a delayed write IO and its associate cache entries not currently in destaging. Even though a few efficient schemes exist to select a delayed write IO to be destaged into a disk, we employ a FIFO-based scheme because this selection is not our concern. Next, it decides if the old data and parity blocks of the entry reside in the read cache. If not, it issues destage read requests for the missing blocks. Note that the old data and parity blocks are read into the read cache, and they are not locked in the read cache until the computation of its new parity blocks is completed. This implies that these blocks can be replaced with others before the new parity is calculated. After reading all the old data and parity blocks into the read cache by the destage reads, it computes the new parity and generates two destage write requests for the new data and parity. Note that newly computed parity information is overwritten into the cache entries of the old parity information in the read cache. However, the new parity information is instantly written into physical disks as soon as the new parity information is computed. When the destage write requests are completed, the state of the dirty cache entry is changed into clean, as previously described.

4. Performance Evaluations

We will begin by describing experimental environments. Next, we will compare the proposed algorithm with the HLWM algorithm with various synthetic and traced workloads.

Table 1 The inter-arrival time and spatial locality for each time-period of W^1_{synth} and W^2_{synth}.

time period	avg. inter-arrival time	% of entire disk to be accessed	associated workload pattern
T_1	20 msec	0.4%	initial
T_2	40 msec	0.4%	Case 1
T_3	20 msec	0.4%	initial
T_4	10 msec	0.4%	Case 2
T_5	20 msec	0.4%	initial
T_6	20 msec	0.2%	Case 3
T_7	20 msec	0.4%	initial
T_8	20 msec	0.8%	Case 4

4.1 Experimental Environments

A RAID5 system, specifically a logical unit, under test consists of eight Seagate Fibre Channel disks. Its space is configured as 10 GB. Its stripe unit size is 16 KB. Each of the read and write caches is configured as 64 MB and its cache entry size is set to 16 KB. That is, $N_W = 4096$. The ρ_H and ρ_L are initially set to 0.5 and 0.1 in an ad-hoc manner, implying that $\mathcal{H}_{static} = 2048$ and $\mathcal{L}_{static} = 410$. Note that these $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ values are not optimal. The values of $i(t)$ and $\delta(t)$ are updated every 3 seconds.

We use two different types of the synthetic workloads which are the read-dominant W^1_{synth} and the write-dominant W^2_{synth}; that is, the read-to-write ratios of W^1_{synth} and W^2_{synth} are 60% and 40%, respectively. The basic access pattern of each workload simulates requests from five processes that access subsets of the entire storage space. A single process uniformly accesses the whole space, and each of the four processes uniformly accesses a certain percentage of the entire storage space. The request size is fixed at 16 KB, which is equal to the cache entry size. The inter-arrival time, which is defined as a time difference between the arrivals of two adjacent requests, is determined randomly from a negative exponential distribution. Each synthetic workload encompasses eight time-periods, each of which is denoted with T_i. Each time-period T_i runs for 5 minutes and specifies an increasing rate of the write cache occupancy level controlled by an average inter-arrival time and a decreasing rate of the write cache occupancy level controlled by a percentage of the entire storage space, called a spatial locality (See Table 1). Notice that the time-periods for T_2, T_4, T_6, and T_8 respectively correspond to Case 1, Case 2, Case 3, and Case 4 in Sect. 2.3.1, and the increasing and decreasing rates of the write cache occupancy level at T_3, and T_5, and T_7 are the same as those at T_1.

In addition, we use two different types of the traced workloads of two hours long from all the traced workloads of the `cello` system, as shown in Table 2. W^1_{traced}

Table 2 IO characteristics of the traced workloads, W^1_{traced} and W^2_{traced}.

type	traced period	read ratio	avg. req. size	avg. inter-arrival time
W^1_{traced}	05/04/92 10am–12am	75.19%	6.5 KB	21.10 msec
W^2_{traced}	06/15/92 10am–12am	17.86%	7.3 KB	34.47 msec

is dominated by read IOs with 170569 IOs and W^2_{traced} is by write IOs with 104453 IOs. We have doubled each inter-arrival rate of the traced workloads. The following adjustments are made for the generation of the traced workloads. First, the eight(8) disks in the `cello` system are disjointly mapped into the logical address space of our RAID architecture with 8 disks. Second, we reduced the original inter-arrival times by half to reflect the increased traffic intensity since the year when the trace was obtained. Finally, we set up a Linux-based host system which regenerates each traced workload via Linux SCSI Generic interface [8] in an asynchronous manner based on its arrival time at its device driver queue. The two synthetic workloads are also generated from this Linux-based host system.

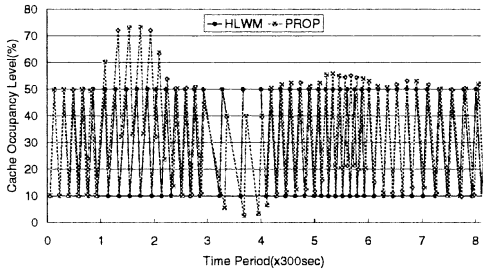
The performance metrics include the read cache hit ratio, the overwrite ratio, and the disk IO traffic which is directly associated with the amount of IO traffic to and/or from physical disks by the destaging process. In addition, we examine the variation of the HWM and LWM values as a function of time with various types of workloads.

4.2 Performance Results

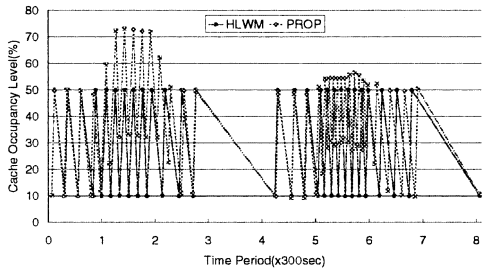
By employing synthetic workloads, we validate the ability of the proposed algorithm to automatically adjust its thresholds for the changed workloads. Next, we additionally use traced workloads to see how effectively the proposed algorithm works under real-world workloads.

4.2.1 Synthetic Workloads

Figure 4 shows the variation of the HWM and LWM values with the synthetic workload W^1_{synth} and W^2_{synth}. At T_1, the proposed algorithm extracts $i_B = 70.09$ and $\delta_B = 59.08$ from W^1_{synth} and $i_B = 104.00$ and $\delta_B = 28.92$ from W^2_{synth}. Note that W^2_{synth} causes heavier traffic toward the write cache, because it is dominated by write IOs. At T_2, $i(t)$ becomes two times slower than i_B due to the increased inter-arrival time (Case 1). The proposed algorithm increases the HWM and the LWM upto 73% and 33%, respectively. Conversely, at T_4, $i(t)$ becomes two times faster than i_B due to the decreased inter-arrival time (Case 2). The proposed algorithm decreases the HWM and the LWM to



(a) Synthetic workload W^1_{synth}.



(b) Synthetic workload W^2_{synth}.

Fig. 4 Variations of the write cache occupancy level of the HLWM algorithm and the proposed algorithm under synthetic workloads: (a) W^1_{synth} (read=60%) and (b) W^2_{synth} (write=60%).

39% and 2% in the case of W^1_{synth}. Since $i(t)$ of W^2_{synth} is not slow enough to decrease the current write cache occupancy level to the current LWM at T_4, the destaging process continues during this period. At T_6, $\delta(t)$ becomes faster than δ_B (Case 3) due to the increased spatial locality. The proposed algorithm increases the LWM upto 21–31% while the HWM remains nearby at 50 percent. Observe that the HWM is also increased slightly because $i(t)$ becomes a bit slower due to the increased spatial locality and overwrite ratio. At T_8, $\delta(t)$ becomes slower than δ_B (Case 4) due to the decreased spatial locality. The proposed algorithm decreases the LWM to 9 percent. Notice that the decreasing amount of the LWM at T_8 is relatively smaller than the increasing amount of the LWM at T_6. In the case of W^2_{synth}, the destaging process continues during the whole period as with T_4. Observe that the HWM and LWM values for the time-periods of T_3, T_5, and T_7 are the same as those at T_1. Finally, the write cache occupancy level of the HLWM algorithm alternates between 50% and 10% according to its initial configuration.

Figure 5 presents the overwrite ratio of the HLWM algorithm and the proposed algorithm. At T_2 (Case 1) and T_6 (Case 3), the overwrite ratio of the proposed algorithm is higher than that of the HLWM algorithm by 56% and 36%, respectively. At T_4 (Case 2), the overwrite ratio of the proposed algorithm becomes worse than the HLWM algorithm, because $\mathcal{H}_{adaptive}(t)$ and $\mathcal{L}_{adaptive}(t)$, respectively, decrease lower than $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$. However, the proposed algorithm is expected to decrease the chances of a write cache miss by

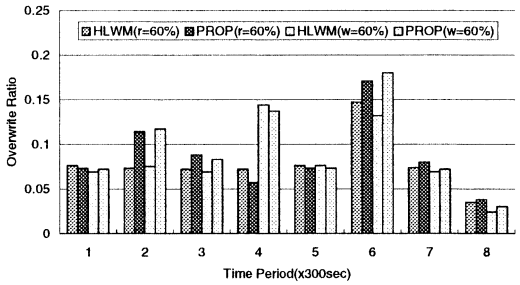


Fig. 5 Overwrite ratio of the HLWM algorithm and the proposed algorithm under synthetic workloads: W^1_{synth} (read=60%) and W^2_{synth} (write=60%).

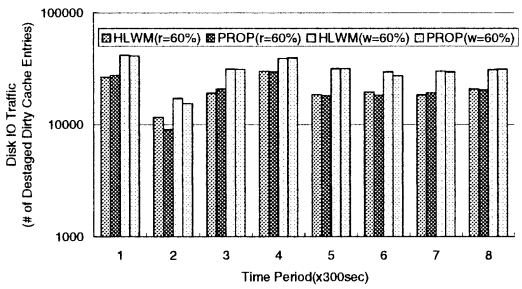


Fig. 6 Disk IO traffic of the HLWM algorithm and the proposed algorithm under the synthetic workloads: W^1_{synth} (read=60%) and W^2_{synth}(write=60%).

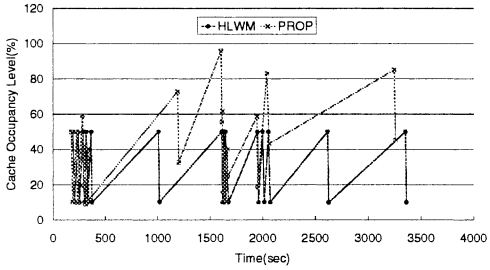
reserving more space to absorb greater write IO traffic. At both T_3 and T_7 which correspond to the succeeding time-periods of T_2 (Case 1) and T_6 (Case 3), the overwrite ratio of the proposed algorithm still remains higher than the HLWM algorithm. By contrast, a lower overwrite ratio continues at T_5. At T_8 (Case 4), the proposed algorithm has a slightly better overwrite ratio, even though its LWM is not higher than $\mathcal{L}_{static}$.

Figure 6 depicts the disk IO traffic of the HLWM algorithm and the proposed algorithm. At T_2 (Case 1) and T_6 (Case 3), the disk IO traffic of the proposed algorithm is better than the HLWM algorithm by 9% and 4%, respectively. At T_3 and T_7 in the transit from Case 1 and Case 3 to the initial workload pattern, the proposed algorithm produces a larger amount of disk IO traffic, as expected in Sect. 2.3.1. At T_4 (Case 2) and T_8 (Case 4), the disk IO traffic of both algorithms are almost the same. The reasons are due to a smaller gain of the overwrite ratio of the HLWM algorithm against the proposed algorithm at T_4 and a relatively lower overwrite ratio at T_8.

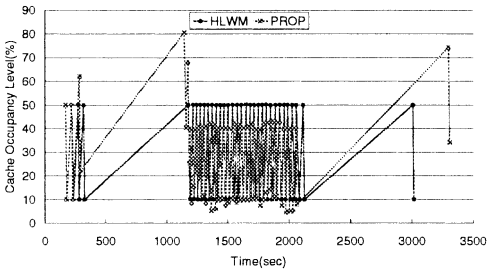
Finally, the read cache hit ratios of both algorithms are observed almost equal, as expected in Sect. 2.3.1. The graphs for the read cache hit ratio are omitted due to space limitations.

4.2.2 Traced Workloads

Figure 7 shows the variation of the HWM and LWM



(a) Traced workload W^1_{traced}.



(b) Traced workload W^2_{traced}.

Fig. 7 Variations of the write cache occupancy level of the HLWM algorithm and the proposed algorithm under the traced workloads of the `cello` system: (a) W^1_{traced} and (b) W^2_{traced}.

values with the read-dominant workload W^1_{traced} and the write-dominant workload W^2_{trace}. Similarly, the proposed algorithm identified the referenced values by sampling nine time-points when the current cache occupancy level reaches either HWM or LWM. Both workloads spent about 4 minutes in order to obtain all the time-points required to compute the referenced values. Note that these values might not fully represent the increasing and decreasing rates of the write cache occupancy levels for the given workload, because the sampling has been made only a small portion of the entire traffic. Measuring more accurate referenced values for a given workload remain for future work. In our current experiment, we measured that W^1_{traced} has $i_B = 53.94$ and $\delta_B = 163.07$, and W^2_{synth} has $i_B = 54.12$ and $\delta_B = 173.86$. Observe that the W^1_{traced} and W^2_{trace} have lighter IO traffic than the synthetic workloads. Also, we will see that the write traffic of both workloads exhibits a high spatial locality. In the case of W^1_{traced}, the proposed algorithm continuously increases the HWM to 72% and 95% and the LWM to 32–55% until 1606 seconds from the start time. Subsequently, it decreases the HWM to 40–39% and the LWM to 24–10 percent. Finally, the HWM and LWM end up being 85% and 45 percent. In the case of W^2_{traced}, the HWM and LWM decrease to about 40% and 10% from 1192 second where $i(t)$ is slightly higher than its base value i_B. Afterwards, it again increases the HWM and LWM to 74% and 34 percent.

Figure 8 gives the overwrite ratio of the HLWM algorithm and the proposed algorithm under the traced workloads. In the case of W^1_{traced}, as shown in Fig. 8,

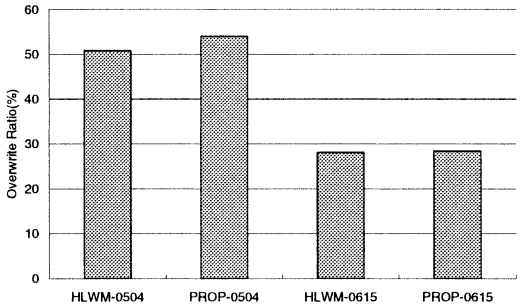


Fig. 8 Overwrite ratio of the HLWM algorithm and the proposed algorithm under traced workloads.

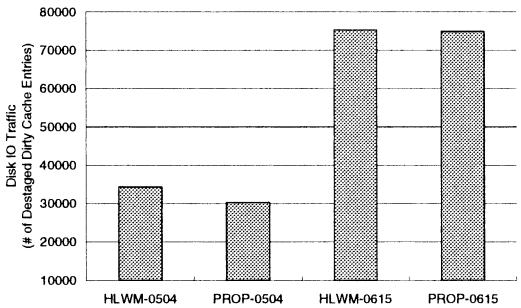


Fig. 9 Disk IO traffic of the HLWM algorithm and the proposed algorithm under the traced workloads: W^1_{traced} and W^2_{traced}.

the overwrite ratio of the proposed algorithm is 6.2% higher than that of the HLWM algorithm, which is 50.8 percent. In the case of W^2_{traced} where the HWM and LWM values of both algorithms are almost equal, the HLWM and proposed algorithms have the same overwrite ratio.

Figure 9 shows the disk IO traffic of the HLWM algorithm and the proposed algorithm. As expected, the proposed algorithm reduced the disk IO traffic by 12% compared with the HLWM algorithm in the case of W^1_{traced}. However, the traffic amount is almost the same in W^2_{traced}.

Finally, the read cache hit ratios of both algorithms are observed to be almost equal to the synthetic workloads. The graphs for the read cache hit ratio are also omitted due to space limitations.

5. Conclusion and Future Work

We addressed the problem of a management overhead that frequently reconfigures the two thresholds of the HLWM algorithm with a changed workload. Next, we proposed an efficient destage algorithm which automatically re-configures its initial thresholds according to changes in traffic intensity and access patterns, called adaptive thresholding. The core of adaptive thresholding is to define two thresholds as the multiplication of the referenced increasing and decreasing rates of the write cache occupancy level and the time required to fill and empty the write cache. We implemented the proposed algorithm on an actual RAID system and then verified its ability to automatically reconfigure its thresholds with synthetic workloads having different traffic intensity and access patterns. Next, performance evaluations under well-known traced workloads revealed that the proposed algorithm reduced the disk IO traffic by about 12% with 6% increase in the overwrite ratio compared with the HLWM algorithm.

In future work, we first need to devise an algorithm which automatically finds the optimal $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ for a given workload. Next, we should devise a scheme to carefully adjust the $\mathcal{H}_{static}$ and $\mathcal{L}_{static}$ not only to avoid any write cache miss, but to maintain the write cache occupancy as high as possible. For this issue, we may look into the problem of tolerating a bursty pattern of IO traffic as well. In addition, we will devise a better scheme to identify the referenced increasing and decreasing rates of the write cache occupancy level in order to represent the average rates more accurately.

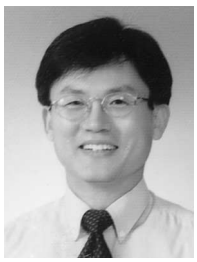
Acknowledgement

The authors would like to thank the Ministry of Education of Korea for its financial support toward the Electrical and Computer Engineering Division at POSTECH through its BK21 program. This research was also supported in part by HY-SDR IT Research Center.

References

[1] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks(RAID)," Proc. ACM SIGMOD, pp.109–116, June 1988.
[2] J. Menon and J. Cortney, "The architecture of a fault-tolerant cached RAID controller," Proc. 20th International Symposium on Computer Architecture, pp.76–86, May 1993.
[3] J. Menon, J. Roche, and J. Kasson, "Floating parity and data disk arrays," J. Parallel Distrib. Comput., pp.129–139, 1993.
[4] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," ACM Trans. Computer Systems, vol.10, pp.206–235, Aug. 1994.
[5] D. Stodolsky, G. Gibson, and M. Holland, "Parity logging: Overcoming the small write problem in disk arrays," Proc. 20th International Symposium on Computer Architecture, May 1993.
[6] A. Varma and Q. Jacobson, "Destage algorithms for disk arrays with nonvolatile caches," IEEE Trans. Comput., vol.47, no.2, pp.228–235, Feb. 1998.
[7] Y. Nam, D. Kim, T. Choe, and C. Park, "Enhancing write I/O performance of disk array RM2 tolerating double disk failures," Proc. 31st International Conference on Parallel and Processing, Aug. 2002.
[8] "Linux scsi generic (sg) howto." URL: http://www.tldp.org/HOWTO/SCSI-Generic-HOWTO/.

Young Jin Nam received B.S. and M.S. degrees in electronics engineering from Kyungpook National University and Pohang University of Science and Technology, Korea in 1992 and 1994, respectively. He joined Electronics and Telecommunications Research Institute, Korea in 1994, where he engaged in the development of a microkernel-based operating system for a massively parallel computer. Since 1998, he has been pursuing a Ph.D. degree in computer science and engineering at Pohang University of Science and Technology, Korea. His research interests include high-performance, highly reliable storage architectures and storage quality of service.



Chanik Park received a B.E. degree in 1983 from Seoul National University, Seoul, Korea, an M.S. degree in 1985, and a Ph.D. degree in 1988, both from Korea Advanced Institute of Science and Technology, Taejeon, Korea. Since 1989, he has been working for Pohang University of Science and Technology, where he is currently an Associate Professor with the Department of Computer Science and Engineering. He was a visiting scholar with Parallel Systems group in the IBM Thomas J. Watson Research Center in 1991, and a visiting professor with Storage Systems group in the IBM Almaden Research Center in 1999. He served a number of international conferences as a member of Program Committee. He is a member of technical committee in the SIGStorage Systems in Korea Information Processing Society. His research interests include storage systems, embedded systems, and pervasive computing.