# Design and Evaluation of a New Approach to RAID-0 Scaling

GUANGYAN ZHANG and WEIMIN ZHENG, Tsinghua University
KEQIN LI, State University of New York and Tsinghua University

Scaling up a RAID-0 volume with added disks can increase its storage capacity and I/O bandwidth simultaneously. For preserving a round-robin data distribution, existing scaling approaches require all the data to be migrated. Such large data migration results in a long redistribution time as well as a negative impact on application performance. In this article, we present a new approach to RAID-0 scaling called FastScale. First, FastScale minimizes data migration, while maintaining a uniform data distribution. It moves only enough data blocks from old disks to fill an appropriate fraction of new disks. Second, FastScale optimizes data migration with access aggregation and lazy checkpoint. Access aggregation enables data migration to have a larger throughput due to a decrement of disk seeks. Lazy checkpoint minimizes the number of metadata writes without compromising data consistency. Using several real system disk traces, we evaluate the performance of FastScale through comparison with SLAS, one of the most efficient existing scaling approaches. The experiments show that FastScale can reduce redistribution time by up to 86.06% with smaller application I/O latencies. The experiments also illustrate that the performance of RAID-0 scaled using FastScale is almost identical to, or even better than, that of the round-robin RAID-0.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms: Algorithms, Design, Experimentation, Management, Performance

Additional Key Words and Phrases: Access aggregation, data migration, lazy checkpoint, RAID scaling

**ACM Reference Format:**
Zhang, G., Zheng, W., and Li, K. 2013. Design and evaluation of a new approach to RAID-0 scaling. *ACM Trans. Storage* 9, 4, Article 11 (November 2013), 31 pages.
DOI:http://dx.doi.org/10.1145/2491054

## 1. INTRODUCTION

### 1.1. Motivation

Redundant Array of Inexpensive Disks (RAID) was proposed to achieve high performance, large capacity, and data reliability, while allowing a RAID volume to be managed as a single device [Patterson et al. 1988]. This goal is achieved via disk striping and rotated parity. RAID has been well studied and widely used in high bandwidth and space-demanding areas. As user data increase and computing power is enhanced, applications often require larger storage capacity and higher I/O throughput. The scalability issue of RAID has become a main concern. To provide the required capacity

This work was supported by the National Natural Science Foundation of China under Grants 60903183, 61170008, and 61272055, the National High Technology Research and Development 863 Program of China under Grant 2013AA01A210, and the National Grand Fundamental Research 973 Program of China under Grant No. 2014CB340402.

Authors' addresses: G. Zhang and W. Zheng, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China; email: {gyzh, zwm-dcs}@tsinghua.edu.cn; K. Li, Department of Computer Science, State University of New York, New Paltz, New York 12561, USA; email: lik@newpaltz.edu.


DOI:http://dx.doi.org/10.1145/2491054

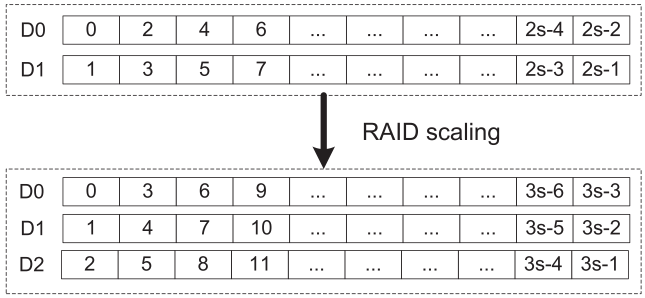



Fig. 1. RAID scaling from 2 disks to 3 using traditional approaches. All data blocks except blocks 0 and 1 have to be migrated.

and/or bandwidth, one solution is to add new disks to a RAID volume. Such disk addition is termed *RAID scaling*.

To regain uniform data distribution in all disks including the old and the new ones, RAID scaling requires certain blocks to be moved onto added disks. Furthermore, in today's server environments, many applications (e.g., e-business, scientific computation, and Web servers) access data constantly. The cost of downtime is extremely high [Patterson 2002], giving rise to the necessity of online and real-time scaling.

Traditional approaches to RAID scaling [Brown 2006; Gonzalez and Cortes 2004; Zhang et al. 2007] are restricted by preserving the round-robin order after adding disks. Let $\mathbb{N}$ be the set of nonnegative integers. The addressing algorithm $f_i(x) : \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ can be expressed as follows for the $i$th scaling operation:

$$f_i(x) = (d, b) = (x \bmod N_i, x/N_i), \tag{1}$$

where block $b = x/N_i$ of disk $d = x \bmod N_i$, is the location of logical block $x$, and $N_i$ gives the total number of disks. As far as RAID scaling from $N_{i-1}$ disks to $N_i$ is concerned, since $N_i \neq N_{i-1}$, we have $f_i(x) \neq f_{i-1}(x)$ for almost every block $x$. Supposing $f_i(x) = f_{i-1}(x)$, we have $x = b \times N_{i-1} + d$ and $x = b \times N_i + d$, which implies that $b \times (N_i - N_{i-1}) = 0$. Since $N_i \neq N_{i-1}$, we have $b = 0$. In other words, only the data blocks in the first stripe ($b = 0$) are not moved. As an example, Figure 1 illustrates the data distributions before and after RAID scaling from 2 disks to 3. We can see that all data blocks except blocks 0 and 1 are moved during this scaling.

Suppose each disk consists of $s$ data blocks. Let $r_1$ be the fraction of data blocks to be migrated. We have

$$r_1 = \frac{N_{i-1} \times s - N_{i-1}}{N_{i-1} \times s} = 1 - \frac{1}{s}. \tag{2}$$

Since $s$ is very large, we have $r_1 \approx 100\%$. This indicates that almost 100 percent of data blocks have to be migrated no matter what the numbers of old disks and new disks are. There have been some efforts concentrating on optimization of data migration [Gonzalez and Cortes 2004; Zhang et al. 2007]. They improve the performance of RAID scaling to a certain extent, but do not completely overcome the limitation of large data migration.

The most intuitive method to reduce data migration is the semi-RR algorithm [Goel et al. 2002]. It requires a block to be moved only if the resulting disk number is one of the new disks. The algorithm can be expressed as follows for the $i$th scaling operation:

$$g_i(x) = \begin{cases} g_{i-1}(x), & \text{if } (x \bmod N_i) < N_{i-1}; \\ f_i(x), & \text{otherwise.} \end{cases} \tag{3}$$

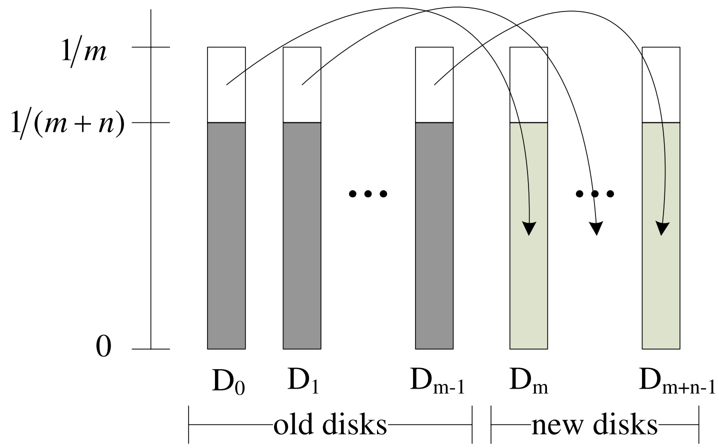



Fig. 2. Data migration using FastScale. A minimum amount of data blocks are moved from old disks to new disks to regain a uniform distribution, while no datum is migrated among old disks.

Semi-RR reduces data migration significantly. Unfortunately, it does not guarantee uniform distribution of data blocks after successive scaling operations (see Section 2.6). This will deteriorate the initial evenly distributed load.

It is clear that it is still an open problem as to whether there exists a RAID scaling method that is able to maintain a uniform and balanced load distribution by performing the minimum amount of data migration.

### 1.2. Our Contributions

In this article, we propose a novel approach called *FastScale* to redistribute data for RAID-0 scaling. We have made three significant contributions in developing FastScale.

The first contribution is that FastScale accelerates RAID-0 scaling by minimizing data migration. As shown in Figure 2, FastScale moves data blocks from old disks to new disks just enough for preserving the uniformity of data distribution, while not migrating data among old disks. Before performing RAID scaling from $m$ disks to $m + n$, the old disks hold $m \times s$ data blocks and the new disks hold no data. After RAID scaling, each disk, either old or new, holds $1/(m + n)$ of the total data to meet the uniform distribution requirement. Without loss of generality, let $r_2$ be the fraction of data blocks to be migrated. We have

$$r_2 = \frac{(N_{i-1} \times s) \times \frac{1}{N_i} \times (N_i - N_{i-1})}{N_{i-1} \times s} = \frac{N_i - N_{i-1}}{N_i}. \tag{4}$$

For instance, for RAID scaling from 3 disks to 4, we have $r_2 = (4 - 3)/4 = 25\%$. To regain a uniform distribution, data migration from old disks to new ones is necessary. Consequently, the migration fraction $r_2$ of FastScale reaches the lower bound $r^*$ of the migration fraction, where $r^* = 1 - N_{i-1}/N_i$. In other words, FastScale succeeds in minimizing data migration for RAID-0 scaling.

We design an elastic addressing function through which the location of one block can be easily computed without any lookup operation. By using this function, FastScale changes only a fraction of the data layout while preserving the uniformity of data distribution. FastScale has several unique features, which are listed as follows.

— FastScale maintains a uniform data distribution after each RAID scaling.
— FastScale minimizes the amount of data to be migrated during each RAID scaling.

— FastScale preserves simple management of data due to deterministic placement.
— FastScale can sustain the three preceding features after multiple disk additions.

The second contribution is that FastScale exploits special physical properties to optimize online data migration. First, it uses aggregated accesses to improve the efficiency of data migration. Second, it records data migration lazily to minimize the number of metadata updates while ensuring data consistency.

The third contribution is that FastScale has significant performance improvement. We implement a detailed simulator that uses DiskSim as a worker module to simulate disk accesses. Under several real system workloads, we evaluate the performance of traditional approaches and the FastScale approach. The experimental results demonstrate the following results.

— Compared with SLAS, one of the most efficient traditional approaches, FastScale shortens redistribution time by up to 86.06% with smaller maximum response time of user I/Os.
— The performance of RAID scaled using FastScale is almost identical to, or even better than, that of the round-robin RAID.

In this article, we only describe our solution for RAID-0, i.e., striping without parity. The solution can also work for RAID-10 and RAID-01. Therefore, FastScale can be used in disk arrays, logical volume managers, and file systems. Although we do not handle RAID-4 and RAID-5, we believe that our method provides a good starting point for efficient scaling of RAID-4 and RAID-5 arrays. We will report this part of our investigation in a future paper.

### 1.3. Differences from Our Prior Work

This article is based on our prior work presented at the 9th USENIX Conference on File and Storage Technologies (FAST'11) [Zheng and Zhang 2011]. In this article, the following new and important materials beyond the earlier version are provided.

— We improve the addressing algorithm of FastScale, especially change the way that a block newly added after the last scaling is placed. The addressing algorithm describes how FastScale maps a logical address of a RAID-0 array to a physical address of a member disk. Furthermore, the presentation of the two examples is also revised accordingly.
— We design the demapping algorithm of FastScale. In many cases, it is also required to map a physical address to a logical address. The demapping algorithm can be used to provide such a mapping. The goal of improving the addressing algorithm is to enable designing the demapping algorithm and to make it simple.
— We formally prove that FastScale satisfies all three requirements for an ideal approach to RAID-0 scaling.
— We also perform more experiments in different cases to make the performance evaluation of FastScale more adequate and more convincing.
— Finally, we add some new materials to make the motivation of our work clearer and to help the reader better understand how FastScale works.

### 1.4. Article Organization

The rest of the article is organized as follows. In Section 2, we formally define the problem to be addressed in the article, give illustrative and motivating examples, develop our addressing and demapping algorithms, and prove the properties of FastScale. In Section 3, we present the optimization techniques used in FastScale, i.e., access aggregation and lazy checkpoint. In Section 4, we demonstrate our experimental results to compare the performance of FastScale with that of the existing solutions. In Section 5, we review related work in scaling deterministic and randomized RAID. We conclude the article in Section 6.

## 2. MINIMIZING DATA MIGRATION

### 2.1. Problem Statement

For disk addition into a RAID-0 array, it is desirable to ensure an even load distribution on all the disks and minimal block movement. Since the location of a block may be changed during a scaling operation, another objective is to quickly compute the current location of a block.

As far as the $i$th RAID scaling operation from $N_{i-1}$ disks to $N_i$ is concerned, suppose that each disk consists of $s$ data blocks. Before this scaling operation, there are $N_{i-1} \times s$ blocks stored on $N_{i-1}$ disks. To achieve these objectives, the following three requirements should be satisfied for RAID scaling.

— *Requirement* 1 (*Uniform Data Distribution*). After this scaling operation, the expected number of data blocks on each one of the $N_i$ disks is $(N_{i-1} \times s)/N_i$, so as to maintain an even load distribution.
— *Requirement* 2 (*Minimal Data Migration*). During this scaling operation, the expected number of data blocks to be moved is $(N_{i-1} \times s) \times (N_i - N_{i-1})/N_i$.
— *Requirement* 3 (*Fast Data Addressing*). After this scaling operation, the location of a block is computed by an algorithm with low space and time complexities for all original and unmoved data, original and migrated data, and new data.

### 2.2. Examples of RAID Scaling Using FastScale

*Example* 1. To understand how the FastScale algorithm works and how it satisfies all three requirements, we take RAID scaling from 3 disks to 5 as an example. As shown in Figure 3, one RAID scaling process can be divided into two stages logically, i.e., data migration and data filling. In the first stage, a fraction of existing data blocks are migrated to new disks. In the second stage, new data are filled into the RAID continuously. Actually, the two stages, data migration and data filling, can be overlapped in time.

For the RAID scaling, each 5 consecutive locations in one disk are grouped into one *segment*. For the 5 disks, 5 segments with the same physical address are grouped into one *region*. Locations on all disks with the same block number form a *column* or a *stripe*. In Figure 3, different regions are separated by wavy lines. For different regions, the ways for data migration and data filling are completely identical. Therefore, we will focus on one region, and let $s = 5$ be the number of data blocks in one segment.

In a region, all data blocks within a parallelogram will be moved. The base of the parallelogram is 2, and the height is 3. In other words, 2 data blocks are selected from each old disk and migrated to new disks. The 2 blocks are sequential—the start address is the disk number *disk_no*. Figure 3 depicts the moving trace of each migrating block. For one moving data block, its physical disk number is changed while its physical block number is unchanged. As a result, the five columns of two new disks will contain 1, 2, 2, 1, and 0 migrated data blocks, respectively. Here, the data block in the first column will be placed on disk 3, while the data block in the fourth column will be placed on disk 4. The first blocks in columns 2 and 3 are placed on disk 3, and the second blocks in columns 2 and 3 are placed on disk 4. Thus, each new disk has 3 data blocks.

After data migration, each disk, either old or new, has 3 data blocks. That is to say, FastScale regains a uniform data distribution. The total number of data blocks to be moved is $2 \times 3 = 6$. This reaches the minimal number of moved blocks in each region,

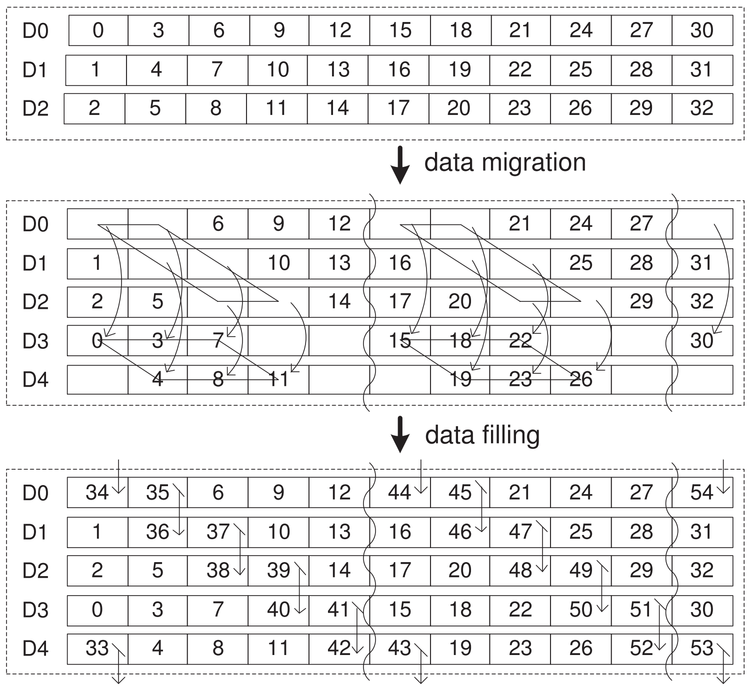



Fig. 3. RAID scaling from 3 disks to 5 using FastScale, where $m \geq n$.

i.e., $(N_{i-1} \times s) \times (N_i - N_{i-1})/N_i = (3 \times 5) \times (5-3)/5 = 6$. We can claim that the RAID scaling using FastScale can satisfy Requirement 1 and Requirement 2.

Let us examine whether FastScale can satisfy Requirement 3, i.e., fast data addressing. To consider how one logical data block is addressed, we divide the data space in the RAID into three categories: original and unmoved data, original and migrated data, and new data. The conclusion can be drawn from the following description that the calculation overhead for the data addressing is very low.

— The original and unmoved data can be addressed with the original addressing method. In this example, the ordinal number of the disk that holds one block $x$ can be calculated as $d = x \bmod 3$. Its physical block number can be calculated as $b = x/3$.
— The addressing method for original and migrated data can be obtained easily from the description about the trace of the data migration. First, we have $b = x/3$. Next, for those blocks in the first triangle, i.e., blocks 0, 3, and 4, we have $d = d_0 + 3$. For those blocks in the last triangle, i.e., blocks 7, 8, and 11, we have $d = d_0 + 2$. Here, $d_0$ is the original disk number of a block.
— Each region can hold $5 \times 2 = 10$ new blocks. In one region, how those new data blocks are placed is shown in Figure 3. If block $x$ is a new block, it is the $y$th new block, where $y = x - 3 \times 11$. Each stripe holds 2 new blocks. So, we have $b = y/2$. For each new data block, we have $d = (((b-1) \bmod 5) + (y \bmod 2)) \bmod 5$.

*Example* 2. In the preceding example, the number of old disks $m$ and the number of new disks $n$ satisfy the condition $m \geq n$. In the following, we examine the case when $m < n$. Take RAID scaling from 2 disks to 5 as an example, where, $m = 2$ and $n = 3$.

Likewise, in a region, all data blocks within a parallelogram will be moved. The base of the parallelogram is 3, and the height is 2. 3 consecutive data blocks are selected from each old disk and migrated to new disks. Figure 4 depicts the trace of each

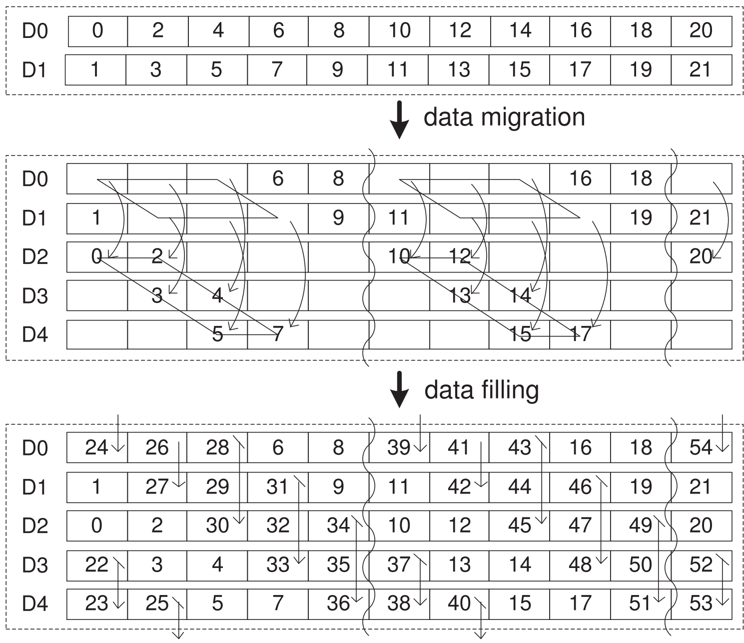



Fig. 4. RAID scaling from 2 disks to 5 using FastScale, where $m < n$.

migrating block. Similarly, for one moving data block, only its physical disk number is changed, while its physical block number is unchanged. As a result, five columns of three new disks will have different numbers of existing data blocks: 1, 2, 2, 1, 0. Here, the data block in the first column will be placed on disk 2, while the data block in the fourth column will be placed on disk 4. Unlike the first example, the first blocks in columns 2 and 3 are placed on disks 2 and 3, respectively. Thus, each new disk has 2 data blocks.

After data migration, each disk, either old or new, has 2 data blocks. That is to say, FastScale regains a uniform data distribution. The total number of data blocks to be moved is $3 \times 2 = 6$. This reaches the minimal number of moved blocks, $(2 \times 5) \times (5 - 2)/5 = 6$. We can claim that the RAID scaling using FastScale can satisfy Requirement 1 and Requirement 2.

Let us examine whether FastScale can satisfy Requirement 3, i.e., fast data addressing. To consider how one logical data block is addressed, we divide the data space in the RAID into three categories: original and unmoved data, original and migrated data, and new data. The conclusion can be drawn from the following description that the calculation overhead for the data addressing is very low.

— The original and unmoved data can be addressed with the original addressing method. In this example, the ordinal number of the disk holds that one block $x$ can be calculated as $d = x \bmod 2$. Its physical block number can be calculated as $b = x/2$.
— The addressing method for original and migrated data can be easily obtained from the description about the trace of the data migration. First, we have $b = x/2$. Next, for those blocks in the first triangle, i.e., blocks 0, 2, and 3, we have $d = d_0 + 2$. For those blocks in the last triangle, i.e., blocks 4, 5, and 7, we have $d = d_0 + 3$. Here, $d_0$ is the original disk number of a block.

— Each region can hold $5 \times 3 = 15$ new blocks. In one region, how those new data blocks are placed is shown in Figure 4. If block $x$ is a new block, it is the $y$th new block, where $y = x - 2 \times 11$. Each stripe holds 3 new blocks. So, we have $b = y/3$. For each new data block, we have $d = (((b - 2) \bmod 5) + (y \bmod 3)) \bmod 5$.

Similar to the first example, we can again claim that the RAID scaling using FastScale can satisfy the three requirements.

### 2.3. The Addressing Algorithm of FastScale

*2.3.1. The Addressing Algorithm.* Figure 5 shows the addressing algorithm for minimizing the data migration required by RAID scaling. An array $N$ is used to record the history of RAID scaling. $N[0]$ is the initial number of disks in the RAID. After the $i$th scaling operation, the RAID consists of $N[i]$ disks.

When a RAID is constructed from scratch ($t = 0$), it is actually a round-robin RAID. The address of block $x$ can be calculated via one division and one modular operations (lines 3–4).

Let us examine the $t$th scaling, where $n$ disks are added into a RAID made up of $m$ disks (lines 7–8).

(1) If block $x$ is an original block (line 9), FastScale calculates its old address $(d_0, b_0)$ before the $t$th scaling (line 10).
   - If the block $(d_0, b_0)$ needs to be moved (line 12), FastScale changes the disk ordinal number via the Moving() function (line 13), while it keeps the block ordinal number unchanged (line 14).
   - If the block $(d_0, b_0)$ does not need to be moved (line 15), FastScale keeps the disk ordinal number and the block ordinal number unchanged (lines 16–17).

(2) If block $x$ is a new block (line 19), FastScale places it via the Placing() procedure (line 20).

*2.3.2. The Moving Function.* The code of line 12 in Figure 5 is used to decide whether a data block $(d_0, b_0)$ will be moved during a RAID scaling. As shown in Figures 3 and 4, there is a parallelogram in each region. The base of the parallelogram is $n$, and the height is $m$. If and only if the data block is within a parallelogram, it will be moved. One parallelogram mapped to disk $d_0$ is a line segment. Its beginning and ending columns are $d_0$ and $d_0 + n - 1$, respectively. If $b_1$ is within the line segment, block $x$ is within the parallelogram, and therefore it will be moved.

Once a data block is determined to be moved, FastScale changes its disk ordinal number with the *Moving()* function given in Figure 6. As shown in Figure 7, a migrating parallelogram is divided into three parts: a head triangle, a body parallelogram, and a tail triangle. How a block moves depends on which part it lies in. No matter which is bigger between $m$ and $n$, the head triangle and the tail triangle keep their shapes unchanged. The head triangle will be moved by $m$ disks (lines 3, 9), while the tail triangle will be moved by $n$ disks (lines 5, 11). However, the body is sensitive to the relationship between $m$ and $n$. The body is twisted from a parallelogram to a rectangle when $m \geq n$ (line 6), and from a rectangle to a parallelogram when $m < n$ (line 12). FastScale keeps the relative locations of all blocks in the same column.

*2.3.3. The Placing Procedure.* When block $x$ is at a location newly added after the last scaling, it is addressed via the *Placing( )* procedure given in Figure 8. If block $x$ is a new block, it is the $y$th new block (line 1). Each stripe holds $n$ new blocks. So we have $b = y/n$ (line 2). In stripe $b$, the first new block is on disk $e$ (line 3). Block $x$ is the $r$th new data block in stripe $b$ (line 4). Therefore, the disk number of block $x$ is

**Algorithm 1**: Addressing($t, N, s, x, d, b$).

**Input**: The input parameters are $t$, $N$, $s$, and $x$, where
$t$: the number of scaling times;
$N$: the scaling history ($N[0]$ , $N[1]$, ..., $N[t]$);
$s$: the number of data blocks in one disk;
$x$: a logical block number.
**Output**: The output data are $d$ and $b$, where
$d$: the disk holding block $x$;
$b$: the physical block number on disk $d$.

```
if (t = 0) then
    m ← N[0]; //the number of initial disks
    d ← x mod m;
    b ← x/m;
    return;
end if;
m ← N[t − 1]; //the number of old disks
n ← N[t] −m; //the number of new disks
if (0 ≤ x ≤ m × s − 1) then //an old data block
    Addressing(t − 1, N, s, x, d0, b0); //find the address (d0, b0) before the tth scaling
    b1 ← b0 mod (m + n);
    if (d0 ≤ b1 ≤ d0 + n − 1) then //an original block to be moved
        d ← Moving(d0, b1, m, n);
        b ← b0;
    else //an original block not to be moved
        d ← d0;
        b ← b0;
    end if;
else //a new data block
    Placing(x, m, n, s, d, b);
end if.
```

Fig. 5. The Addressing algorithm used in FastScale.

$(e + r)$ mod $(m + n)$ (line 5). The order of placing new blocks is shown in Figures 3 and 4.

The addressing algorithm of FastScale is very simple and elegant. It requires fewer than 50 lines of C code, reducing the likelihood that a bug will cause a data block to be mapped to a wrong location.

### 2.4. The Demapping Algorithm of FastScale

*2.4.1. The Demapping Algorithm.* The *Addressing* algorithm describes how FastScale maps a logical address of a RAID-0 array to a physical address of a member disk. In many cases, it is also required to map a physical address to a logical address. FastScale uses the *Demapping* algorithm, shown in Figure 9, to provide a mapping from physical addresses to logical addresses. The array $N$ records the history of RAID scaling. $N[0]$ is the initial number of disks in the RAID. After the $i$th scaling operation, the RAID consists of $N[i]$ disks.

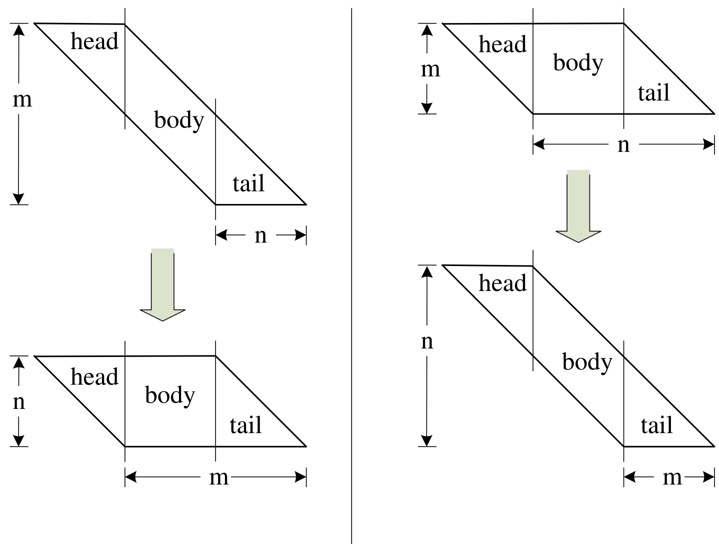
When a RAID is constructed from scratch ($t = 0$), it is actually a round-robin RAID. The logical address of the block at $(d, b)$ can be calculated via one multiplication and one addition operation (line 3).

**Algorithm 2**: Moving($d_0, b_1, m, n$).

**Input**: The input parameters are $d_0$, $b_1$, $m$, and $n$, where
  $d_0$: the disk number before a block is moved;
  $b_1$: the location in a region before a block is moved;
  $m$: the number of old disks;
  $n$: the number of new disks.
**Output**: The disk number after a block is moved.

**if** $(m \geq n)$ **then** (1)
  **if** $(b_1 \leq n - 1)$ **then** //head (2)
    **return** $d_0 + m$; (3)
  **if** $(b_1 \geq m - 1)$ **then** //tail (4)
    **return** $d_0 + n$; (5)
  **return** $m + n - 1 - (b_1 - d_0)$; //body (6)
**else** (7)
  **if** $(b_1 \leq m - 1)$ **then** //head (8)
    **return** $d_0 + m$; (9)
  **if** $(b_1 \geq n - 1)$ **then** //tail (10)
    **return** $d_0 + n$; (11)
  **return** $d_0 + b_1 + 1$; //body (12)
**end if**. (13)

Fig. 6. The Moving function used in the Addressing algorithm.




(a) m⩾n (b) m<n

Fig. 7. The variation of data layout involved in migration.

Let us examine the $t$th scaling, where $n$ disks are added into a RAID made up of $m$ disks (lines 6–7). The logical address of block $(d, b)$ can be calculated in a recursive manner.

— If block $(d, b)$ is an original block and is not moved (line 9), the logical address of block $(d, b)$ remains unchanged across the $t$th scaling (line 10).

**Algorithm 3**: Placing($x, m, n, s, d, b$).

**Input**: The input parameters are $x$, $m$, $n$, and $s$, where
$x$: a logical block number;
$m$: the number of old disks;
$n$: the number of new disks;
$s$: the number of data blocks in one disk.
**Output**: The output data are $d$ and $b$, where
$d$: the disk holding block $x$;
$b$: the physical block number on disk $d$.

$y \leftarrow x - m \times s$;
$b \leftarrow y/n$;
$e \leftarrow (b - (n-1)) \bmod (m+n)$;
$r \leftarrow y \bmod n$;
$d \leftarrow (e + r) \bmod (m+n)$.

Fig. 8. The Placing procedure used in the Addressing algorithm.

**Algorithm 4**: Demapping($t, N, s, d, b$).

**Input**: The input parameters are $t$, $N$, $s$, $d$ and $b$, where
$t$: the number of scaling times;
$N$: the scaling history ($N[0]$ , $N[1]$, ..., $N[t]$);
$s$: the number of data blocks in one disk;
$d$: the disk holding block $x$;
$b$: the physical block number on disk $d$.
**Output**: The output datum is data block $x$ at location $b$ of disk $d$.

(1) **if** $(t = 0)$ **then**
(2) $m \leftarrow N[0]$; //the number of initial disks
(3) $x \leftarrow b \times m + d$;
(4) **return** $x$;
(5) **end if**;
(6) $m \leftarrow N[t-1]$; //the number of old disks
(7) $n \leftarrow N[t] - m$; //the number of new disks
(8) $b_1 \leftarrow b \bmod (m+n)$;
(9) **if** $(0 \le d < m)$ **and** ($b_1 < d$ or $b_1 > d + n - 1$) **then** //an old and unmoved data block
(10) **return** Demapping($t-1, N, s, d, b$);
(11) **end if**;
(12) **if** $(d \ge m)$ **and** $(d - m \le b_1 \le d - 1)$ **then** //an old and moved data block
(13) $d_0 \leftarrow$ Demoving($d, b_1, m, n$); //($d_0, b$) is its location before the $t$th scaling
(14) **return** Demapping($t-1, N, s, d_0, b$);
(15) **end if**;
(16) **return** Deplacing($m, n, s, d, b$). //a new data block

Fig. 9. The Demapping algorithm used in FastScale.

— If block $(d, b)$ is an original block and is moved (line 12), FastScale gets its original location $(d_0, b)$ before the $t$th scaling via the *Demoving( )* function (line 13). It should be remembered that FastScale changes the disk ordinal number while keeping the block ordinal number unchanged. Then, FastScale calculates the logical address of block $(d_0, b)$ before the $t$th scaling (line 14).

**Algorithm 5**: Demoving($d, b_1, m, n$).

**Input**: The input parameters are $d$, $b_1$, $m$, and $n$, where
  $d$: the disk number after a block is moved;
  $b_1$: the location in a region of the block;
  $m$: the number of old disks;
  $n$: the number of new disks.
**Output**: The disk number before a block is moved.

**if** $(m \geq n)$ **then** (1)
  **if** $(b_1 \leq n - 1)$ **then** //head (2)
    **return** $d - m$; (3)
  **if** $(b_1 \geq m - 1)$ **then** //tail (4)
    **return** $d - n$; (5)
  **return** $d + b_1 + 1 - m - n$; //body (6)
**else** (7)
  **if** $(b_1 \leq m - 1)$ **then** //head (8)
    **return** $d - m$; (9)
  **if** $(b_1 \geq n - 1)$ **then** //tail (10)
    **return** $d - n$; (11)
  **return** $d - b_1 - 1$; //body (12)
**end if**. (13)

Fig. 10. The Demoving function used in the Demapping algorithm.

— If block $(d, b)$ is a new block, FastScale gets its logical address via the *Deplacing()* function (line 16).

The code of line 9 is used to decide whether a data block $(d, b)$ is an old block and is not moved during this scaling. As shown in Figures 3 and 4, there is a source parallelogram in each region. The base of the parallelogram is $n$, and the height is $m$. A data block is moved if and only if it is within a parallelogram. One parallelogram mapped to disk $d$ is a line segment. Its beginning and ending columns are $d$ and $d + n - 1$, respectively. If $b_1$ is outside the line segment, block $(d, b)$ is outside the parallelogram, and therefore it is not moved.

*2.4.2. The Demoving Function.* Likewise, the code of line 12 in Figure 9 is used to determine whether a data block $(d, b)$ is an old block and has been moved during this scaling. As shown in Figures 3 and 4, there is a destination parallelogram in each region. The base of the parallelogram is $m$, and the height is $n$. A data block is moved if and only if it is within a destination parallelogram. One parallelogram mapped to disk $d$ is a line segment. Its beginning and ending columns are $d-m$ and $d-1$, respectively. If $b_1$ is within the line segment, block $(d, b)$ is within the parallelogram, and therefore it has been moved.

Once it is determined that a data block has been moved, FastScale gets its original disk number via the *Demoving( )* function given in Figure 10. As shown in Figure 7, a migrating parallelogram is divided into three parts: a head triangle, a body parallelogram, and a tail triangle. How a block moves depends on which part it lies in. No matter which is bigger between $m$ and $n$, the head triangle and the tail triangle keep their shapes unchanged. The head triangle will be moved by $m$ disks (lines 3, 9), while the tail triangle will be moved by $n$ disks (lines 5, 11). However, the body is sensitive to the relationship between $m$ and $n$. The body is twisted from a parallelogram to a rectangle when $m \geq n$ (line 6), while from a rectangle to a parallelogram when $m < n$ (line 12). FastScale keeps the relative locations of all blocks in the same column.

**Algorithm 6**: Deplacing($m, n, s, d, b$).

**Input**: The input parameters are $m$, $n$, $s$, $d$, and $b$, where
$m$: the number of old disks;
$n$: the number of new disks;
$s$: the number of data blocks in one disk.
$d$: the disk holding block $x$;
$b$: the physical block number on disk $d$.
**Output**: The output datum are the logical block number $x$.

$e \leftarrow (b - (n-1)) \bmod (m+n)$; (1)
$r \leftarrow (d - e) \bmod (m+n)$; (2)
$y \leftarrow b \times n + r$; (3)
$x \leftarrow m \times s + y$; (4)
**return** $x$. (5)

Fig. 11. The Deplacing function used in the Demapping algorithm.

*2.4.3. The Deplacing Function.* If block $(d, b)$ is at a location newly added after the $t$th scaling, it is addressed via the *Deplacing( )* function given in Figure 11. Each stripe holds $n$ new blocks. In stripe $b$, the first new block is on disk $e$ (line 1). Therefore, block $(d, b)$ is the $r$th new block in stripe $b$ (line 2). Since each stripe holds $n$ new blocks, block $(d, b)$ is the $y$th new data block (line 3). Furthermore, there are $m \times s$ old data blocks, and therefore, the logical address of block $(d, b)$ is the $x$ (line 4).

### 2.5. Properties of FastScale

In this section, we formally prove that FastScale satisfies all three requirements given in Section 2.1.

THEOREM 2.1. *FastScale maintains a uniform data distribution after each RAID scaling.*

PROOF. Assume that there are $N_{i-1}$ old disks and $N_i - N_{i-1}$ new disks during a RAID scaling. Since each disk is divided into segments of length $N_i$, and a RAID volume is divided into regions with the size of $N_i \times N_i$ locations, it suffices to show that FastScale maintains a uniform data distribution in each region after each RAID scaling. Before a RAID scaling, there are $N_i \times N_{i-1}$ blocks of data on the $N_{i-1}$ old disks. It is clear from Figure 7 that after the scaling, each new disk holds $N_{i-1}$ blocks of data. Since each old disk contributes $N_i - N_{i-1}$ blocks to the new disks, each old disk also holds $N_{i-1}$ blocks of data after the scaling. Hence, the $N_i \times N_{i-1}$ data blocks in a region are evenly distributed over the $N_i$ disks, such that each disk has $N_{i-1}$ blocks in a region. □

THEOREM 2.2. *FastScale performs the minimum number of data migrations during each RAID scaling.*

PROOF. Again, it suffices to show that FastScale performs the minimum number of data migrations in each region during each RAID scaling. According to Requirement 2, the minimum number of blocks to be moved is $(N_{i-1} \times s) \times (N_i - N_{i-1})/N_i$, where each old disk has $s$ data blocks. For one region, each segment on an old disk has $N_i$ data blocks. Therefore, the minimum number of blocks to be moved for one region is $(N_{i-1} \times N_i) \times (N_i - N_{i-1})/N_i = N_{i-1} \times (N_i - N_{i-1})$, which is exactly the number of blocks in one moving parallelogram. □

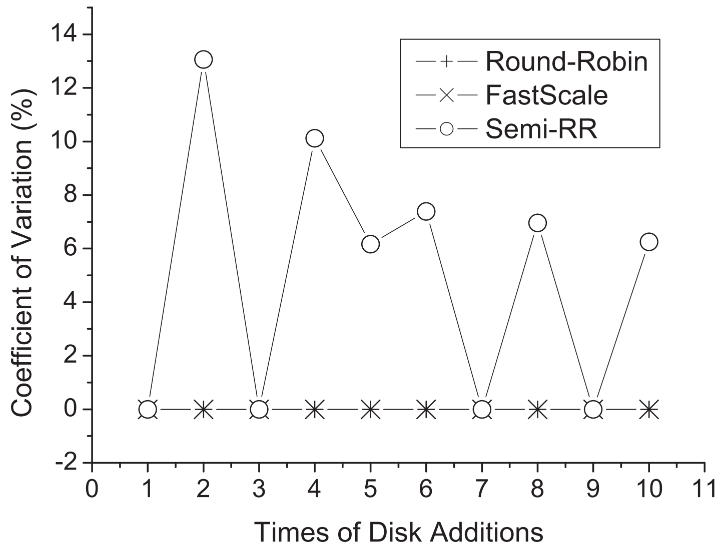



Fig. 12. Comparison of uniformity of data distributions.

THEOREM 2.3. *The Addressing algorithm of FastScale has time complexity $O(t)$ after t RAID scalings.*

PROOF. Let $T(t)$ denote the time complexity of the Addressing algorithm. Since Addressing is a recursive algorithm, we can represent $T(t)$ by using a recurrence relation. First, it is clear that $T(0) = c_1$ for some constant $c_1$. Next, we notice that both the Moving function and the Placing procedure take constant time. Thus, we have $T(t) \leq T(t-1) + c_2$, for all $t \geq 1$, where $c_2$ is some constant. Solving the recurrence relation, we get $T(t) \leq c_1 + c_2 t = O(t)$. □

### 2.6. Property Examination

The purpose of this experiment is to quantitatively characterize whether the FastScale algorithm satisfies the three requirements described in Section 2.1. For this purpose, we compare FastScale with the round-robin algorithm and the semi-RR algorithm. From a 4-disk array, we add one disk repeatedly for 10 times using the three algorithms respectively. Each disk has a capacity of 128 GB, and the size of a data block is 64 KB. In other words, each disk holds $2 \times 1024^2$ blocks.

*2.6.1. Uniform Data Distribution.* We use the coefficient of variation of the numbers of blocks on the disks as a metric to evaluate the uniformity of data distribution across all the disks. The coefficient of variation expresses the standard deviation as a percentage of the average. The smaller the coefficient of variation, the more uniform the data distribution. Figure 12 plots the coefficient of variation versus the number of scaling operations. For the round-robin and FastScale algorithms, both the coefficients of variation remain at 0 percent as the times of disk additions increase.

Conversely, the semi-RR algorithm causes excessive oscillation in the coefficient of variation. The maximum is even 13.06 percent. The reason for this nonuniformity is given as follows. An initial group of 4 disks causes the blocks to be placed in a round-robin fashion. When the first scaling operation adds one disk, 1/5 of all blocks, where ($x$ mod 5) = 4, are moved onto the new disk, i.e., disk 4. However, with another operation of adding one more disk using the same approach, 1/6 of all the blocks are not

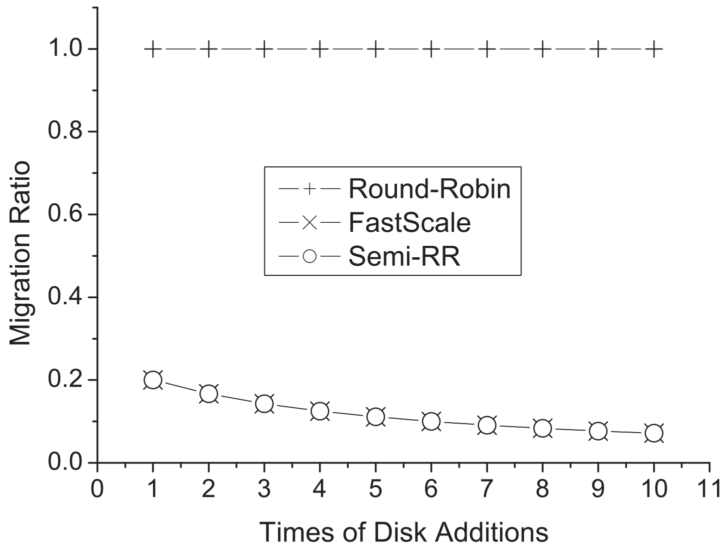



Fig. 13. Comparison of data migration ratios.

Table I. The Storage Overheads of Different Algorithms

| Algorithm | Storage Overhead |
|---|---|
| Round-Robin | 1 |
| Semi-RR | $t$ |
| FastScale | $t$ |

evenly picked from the 5 old disks and moved onto the new disk, i.e., disk 5. Only certain blocks from disks 1, 3, and 4 are moved onto disk 5 while disk 0 and disk 2 are ignored. This is because disk 5 will contain blocks with logical numbers that satisfy $(x \bmod 6) = 5$, which are all odd numbers. The logical numbers of those blocks on disks 0 and 2, resulting from $(x \bmod 4) = 0$ and $(x \bmod 4) = 2$ respectively, are all even numbers. Therefore, blocks from disks 0 and 2 do not qualify and are not moved.

*2.6.2. Minimal Data Migration.* Figure 13 plots the migration fraction (the fraction of data blocks to be migrated) versus the number of scaling operations. Using the round-robin algorithm, the migration fraction is constantly 100%. This will incur a very large migration cost.

The migration fractions using the semi-RR algorithm and using FastScale are identical. They are significantly smaller than the migration fraction of using the round-robin algorithm. Another obvious phenomenon is that they decrease with the increase of the number of scaling operations. The reason for this phenomenon is described as follows. To make each new disk hold $1/N_i$ of total data, the semi-RR algorithm and FastScale move $(N_i - N_{i-1})/N_i$ of total data. $N_i$ increases with the number of scaling operations, i.e., $i$. As a result, the percentage of new disks $((N_i - N_{i-1})/N_i)$ decreases. Therefore, the migration fractions using the semi-RR algorithm and FastScale decrease.

*2.6.3. Storage and Calculation Overheads.* When a disk array boots, it needs to obtain the RAID topology from disks. Table I shows the storage overheads of the three algorithms. The round-robin algorithm depends only on the total number of member disks. So its storage overhead is one integer. The semi-RR and FastScale algorithms depend on how many disks are added during each scaling operation. If we scale RAID $t$ times,

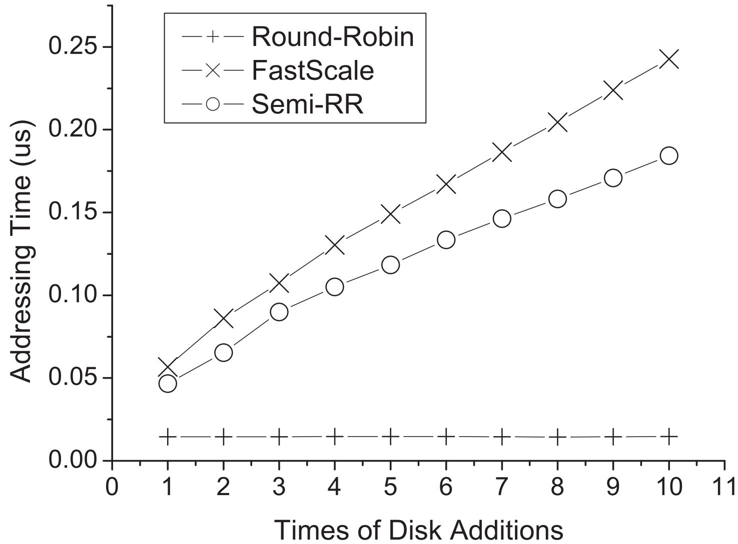



Fig. 14. Comparison of addressing times.

their storage overheads are $t$ integers. Actually, the RAID scaling operation is not too frequent. It may be performed once every half a year, or even less often. Consequently, the storage overheads are very small.

To quantitatively characterize the calculation overheads, we run different algorithms to calculate the physical addresses for all data blocks on a scaled RAID. The whole addressing process is timed and then the average addressing time for each block is calculated. The testbed used in the experiment is an Intel Dual Core T9400 2.53 GHz machine with 4 GB of memory. A Windows 7 Enterprise Edition is installed. Figure 14 plots the addressing time versus the number of scaling operations.

The round-robin algorithm has a low calculation overhead of 0.014 $\mu$s or so. The calculation overheads using the semi-RR and FastScale algorithms are close, and both take on an upward trend. Among the three algorithms, FastScale has the largest overhead. Fortunately, the largest addressing time using FastScale is 0.24 $\mu$s which is negligible compared to milliseconds of disk I/O time.

## 3. OPTIMIZING DATA MIGRATION

The FastScale algorithm succeeds in minimizing data migration for RAID scaling. In this section, we describe FastScale's optimizations for the process of data migration. To better understand how these optimizations work, we first give an overview of the data migration process.

### 3.1. Overview of the Scaling Process

Before the $i$th scaling operation, an addressing equation, $h_{i-1}(x)$, describes the original geometry where $N_{i-1}$ disks serve user requests.

Figure 15 illustrates an overview of the migration process. FastScale uses a sliding window to describe the mapping information of a continuous segment in a RAID under scaling. During scaling, only data that lie within the sliding window is copied to new locations. The addressing information of the sliding window is maintained with a bitmap table, where a bit indicates whether a data block has been migrated. The size of a sliding window is exactly that of a *region*.

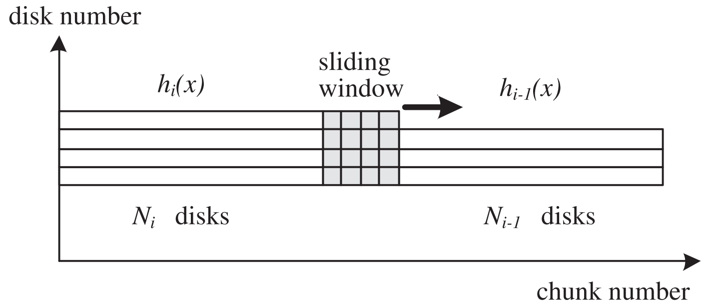



Fig. 15. An overview of the data redistribution process. As the sliding window slides ahead, the newly added disks are gradually available to serve user requests.

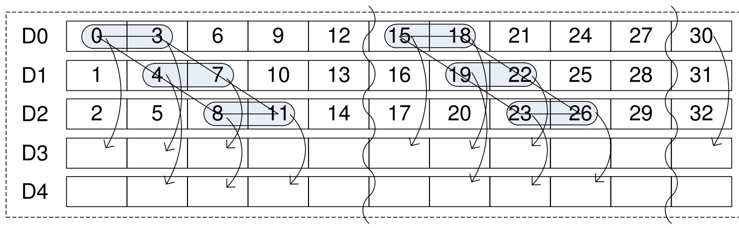



Fig. 16. Aggregate reads for RAID scaling from 3 disks to 5. Multiple successive blocks are read via a single I/O.

An incoming user request is mapped in one of three ways according to its logical address.

— If its logical address is above the sliding window, it is mapped through the equation $h_{i-1}(x)$, where $N_{i-1}$ disks serve user requests.
— If its logical address is below the sliding window, it is mapped through the new equation $h_i(x)$, where $N_i$ disks serve user requests.
— If its logical address is in the range of the sliding window, it is mapped through the sliding window.

When all the data in a sliding window are moved, the sliding window moves ahead by one *region* size. In this way, the newly added disks are gradually available to serve user requests. When data redistribution is completed, the new addressing equation, $h_i(x)$, is used to describe the new geometry.

### 3.2. Access Aggregation

FastScale only moves data blocks from old disks to new disks, while not migrating data among old disks. The data migration will not overwrite any valid data. As a result, data blocks may be moved in an arbitrary order. Since disk I/O performs much better for large sequential accesses, FastScale accesses multiple successive blocks via a single I/O.

Take a RAID scaling from 3 disks to 5 as an example (see Figure 16). Let us focus on the first region. FastScale issues the first I/O request to read blocks 0 and 3, the second request to read blocks 4 and 7, and the third request for blocks 8 and 11, simultaneously. By this means, to read all these blocks, FastScale requires only three I/Os, instead of six. Furthermore, all of these 3 large-size data reads are on three disks. They can be done in parallel, further increasing I/O rate.

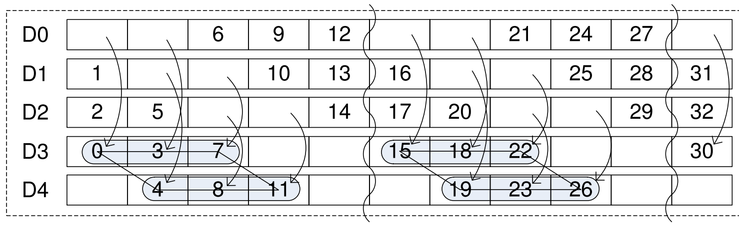



Fig. 17. Aggregate writes for RAID scaling from 3 disks to 5. Multiple successive blocks are written via a single I/O.

When all six blocks have been read into a memory buffer, FastScale issues the first I/O request to write blocks 0, 3, and 7, and the second I/O to write blocks 4, 8, and 11, simultaneously (see Figure 17). In this way, only two large sequential write requests are issued, as opposed to six small writes.

For RAID scaling from $N_{i-1}$ disks to $N_i$ disks, $N_{i-1}$ reads and $N_i - N_{i-1}$ writes are required to migrate all the data in a region, i.e., $N_{i-1} \times (N_i - N_{i-1})$ data blocks.

Access aggregation converts sequences of small requests into fewer, larger requests. As a result, seek cost is mitigated over multiple blocks. Moreover, a typical choice of the optimal block size for RAID is 32KB or 64KB [Brown 2006; Hennessy and Patterson 2003; Kim et al. 2001; Wilkes et al. 1996]. Thus, accessing multiple successive blocks via a single I/O enables FastScale to have a larger throughput. Since data densities in disks increase at a much faster rate than improvements in seek times and rotational speeds, access aggregation benefits more as technology advances.

### 3.3. Lazy Checkpoint

While data migration is in progress, the RAID storage serves user requests. Furthermore, the coming user I/Os may be write requests to migrated data. As a result, if mapping metadata does not get updated until all blocks have been moved, data consistency may be destroyed. Ordered operations [Kim et al. 2001] of copying a data block and updating the mapping metadata (a.k.a., *checkpoint*) can ensure data consistency. But ordered operations cause each block movement to require one metadata write, which results in a large cost for data migration. Because metadata is usually stored at the beginning of all member disks, each metadata update causes one long seek per disk. FastScale uses lazy checkpoint to minimize the number of metadata writes without compromising data consistency.

The foundation of lazy checkpoint is described as follows. Since block copying does not overwrite any valid data, both its new replica and the original are valid after a data block is copied. In the preceding example, we suppose that blocks 0, 3, 4, 7, 8, and 11 have been copied to their new locations and the mapping metadata has not been updated (see Figure 18), when the system fails. The original replicas of the six blocks will be used after the system reboots. As long as blocks 0, 3, 4, 7, 8, and 11 have not been written since being copied, the data remain consistent. Generally speaking, when the mapping information is not updated immediately after a data block is copied, an unexpected system failure only wastes some data accesses, but does not sacrifice data reliability. The only threat is the incoming of write operations to migrated data.

The key idea behind lazy checkpoint is that data blocks are copied to new locations continuously, while the mapping metadata is not updated onto the disks (a.k.a., *checkpoint*) until a threat to data consistency appears. We use $h_i(x)$ to describe the geometry after the $i$th scaling operation, where $N_i$ disks serve user requests. Figure 19 illustrates an overview of the migration process. Data in the moving region is copied to new locations. When a user request arrives, if its physical block address is above the

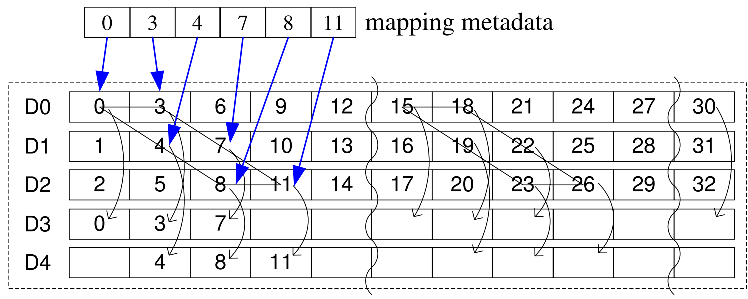



Fig. 18. If data blocks are copied to their new locations and metadata is not yet updated when the system fails, data consistency is still maintained because the data in their original locations are valid and available.

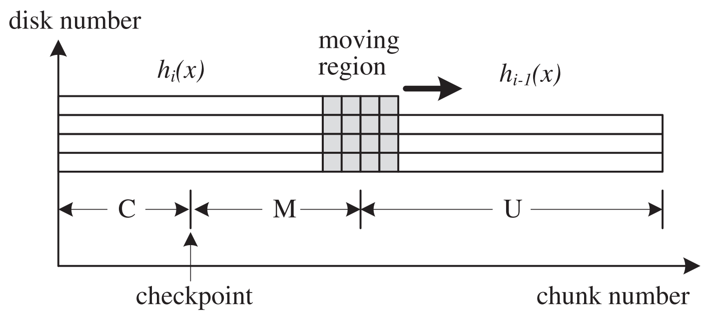



Fig. 19. Lazy updates of mapping metadata. “C”: migrated and checkpointed; “M”: migrated but not checkpointed; “U”:not migrated. Data redistribution is checkpointed only when a user write request arrives in area “M”.

moving region, it is mapped with $h_{i-1}(x)$; If its physical block address is below the moving region, it is mapped with $h_i(x)$. When all of the data in the current moving region are moved, the next region becomes the moving region. In this way, the newly added disks are gradually available to serve user requests. Only when a user write request arrives in the area where data have been moved and the movement has not been checkpointed, are mapping metadata updated.

Since one write of metadata can store multiple map changes of data blocks, lazy updates can significantly decrease the number of metadata updates, reducing the cost of data migration. Furthermore, lazy checkpoint can guarantee data consistency. Even if the system fails unexpectedly, only some data accesses are wasted. It should also be noted that the probability of a system failure is very low.

## 4. EXPERIMENTAL EVALUATION

The experimental results in Section 2.6 show that the semi-RR algorithm causes extremely nonuniform data distribution. This will result in low I/O performance due to load imbalance. In this section, we compare FastScale with the SLAS approach [Zhang et al. 2007] through detailed experiments. SLAS, proposed in 2007, preserves the round-robin order after adding disks.

### 4.1. Simulation System

We use detailed simulations with several disk traces collected in real systems. The simulator is made up of a workload generator and a disk array (Figure 20). According to trace files, the workload generator initiates an I/O request at the appropriate time, so that a particular workload is induced on the disk array.

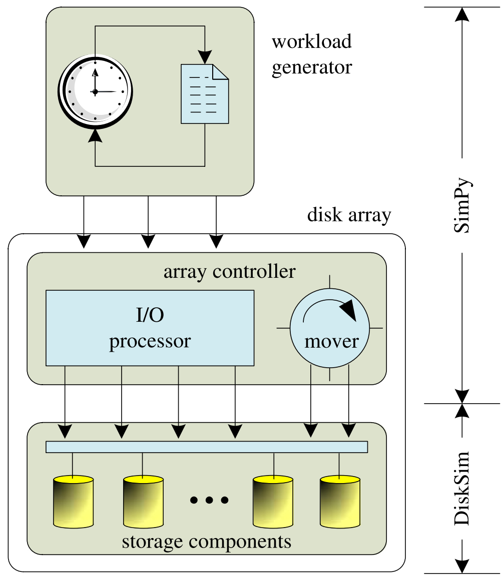



Fig. 20. A simulation system block diagram. The workload generator and the array controller were implemented in SimPy. DiskSim was used as a worker module to simulate disk accesses.

The disk array consists of an array controller and storage components. The array controller is logically divided into two parts: an I/O processor and a data mover. The I/O processor, according to the address mapping, forwards incoming I/O requests to the corresponding disks. The data mover reorganizes the data on the array. The mover uses on/off logic to adjust the redistribution rate. Data redistribution is throttled on detection of high application workload. Otherwise, it performs continuously. An IOPS (I/Os per second) threshold is used to determine whether an application workload is high.

The simulator is implemented in SimPy [Muller and Vignaux 2009] and DiskSim [Bucy et al. 2008]. SimPy is an object-oriented, process-based discrete-event simulation language based on standard Python. DiskSim is an efficient and accurate disk system simulator from Carnegie Mellon University and has been extensively used in various research projects studying storage subsystem architectures. The workload generator and the array controller are implemented in SimPy. Storage components are implemented in DiskSim. In other words, DiskSim is used as a worker module to simulate disk accesses. The simulated disk specification is that of the 15,000-RPM IBM Ultrastar 36Z15 [Hitachi 2001].

### 4.2. Workloads

Our experiments use the following three real system disk I/O traces with different characteristics.

— *TPC-C* traced disk accesses of the TPC-C database benchmark with 20 warehouses [Brigham Young University, 2010.]. It was collected with one client running 20 iterations.
— *Fin* is obtained from the Storage Performance Council (SPC) [UMass Trace Repository 2007, Storage Performance Council 2010], a vendor-neutral standards body. The Fin trace was collected from OLTP applications running at a large financial institution. The write ratio is high.

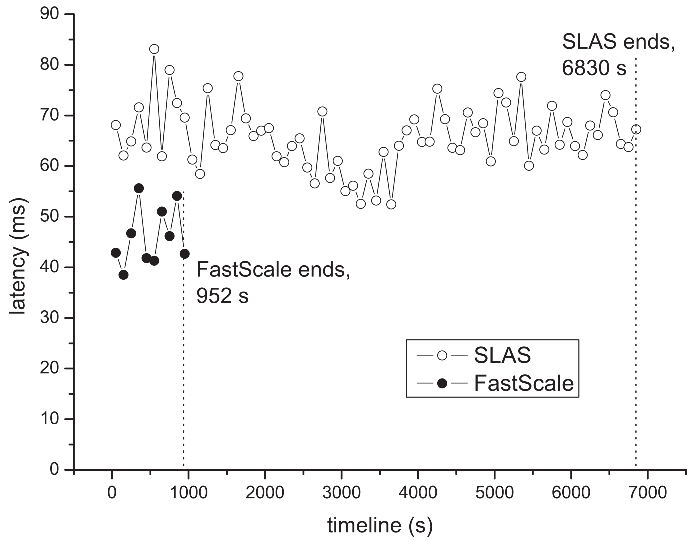



Fig. 21. Performance comparison between FastScale and SLAS under the Fin workload.

— *Web* is also from SPC. It was collected from a system running a Web search engine. The read-dominated Web trace exhibits strong locality in its access pattern.

### 4.3. Experiment Results

*4.3.1. The Scaling Efficiency.* Each experiment lasts from the beginning to the end of data redistribution for RAID scaling. We focus on comparing redistribution times and user I/O latencies when different scaling programs are running in the background.

In all experiments, the sliding window size for SLAS is set to 1024. Access aggregation in SLAS can improve the redistribution efficiency. However, a too-large size of redistribution I/Os will compromise the I/O performance of applications. In our experiments, SLAS reads 8 data blocks via an I/O request.

The purpose of our first experiment is to quantitatively characterize the advantages of FastScale through a comparison with SLAS. We conduct a scaling operation of adding 2 disks to a 4-disk RAID, where each disk has a capacity of 4 GB. Each approach performs with the 32KB stripe unit size under a Fin workload. The threshold of rate control is set to 100 IOPS. This parameter setup acts as the baseline for the latter experiments, from which any change will be stated explicitly.

We collect the latencies of all user I/Os. We divide the I/O latency sequence into multiple sections according to I/O issuing time. The time period of each section is 100 seconds. Furthermore, we get a local maximum latency from each section. A local maximum latency is the maximum of I/O latency in a section. Figure 21 plots local maximum latencies using the two approaches as the time increases along the $x$-axis. It illustrates that FastScale demonstrates a noticeable improvement over SLAS in two metrics.

— First, the redistribution time using FastScale is significantly shorter than that using SLAS: 952 seconds and 6830 seconds, respectively. In other words, FastScale has an 86.06% shorter redistribution time than SLAS. The main factor in FastScale's reduction of the redistribution time is the significant decline of the amount of the data to be moved. When SLAS is used, almost 100% of the data blocks have to be migrated. However, when FastScale is used, only 33.3% of the

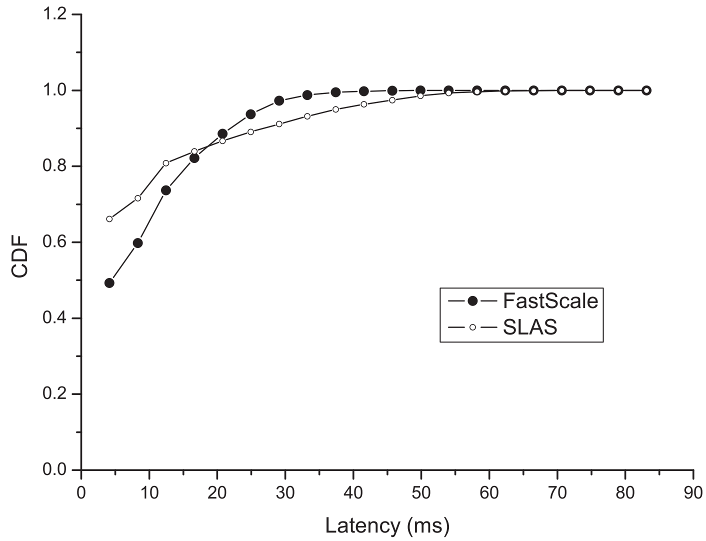



Fig. 22. Cumulative distribution of I/O latencies during data redistributions by the two approaches under the Fin workload.

data blocks have to be migrated. Another factor is the effective exploitation of two optimization technologies: access aggregation reduces the number of redistribution I/Os and lazy checkpoint minimizes metadata writes.

— Second, local maximum latencies of SLAS are obviously longer than those of FastScale. The global maximum latency using SLAS reaches 83.12 ms while that using FastScale is 55.60 ms. This is because the redistribution I/O size using SLAS is larger than that using FastScale. For SLAS, the read size is 256 KB (8 blocks), and the write size is 192 KB (6 blocks). For FastScale, the read size is 64 KB (2 blocks), and the write size is 128 KB (4 blocks). Of course, local maximum latencies of SLAS will be lower with a decrease in the redistribution I/O size. But the decrease in the I/O size will necessarily enlarge the redistribution time.

Figure 22 shows the cumulative distribution of user response times during data redistribution. To provide a fair comparison, I/Os involved in statistics for SLAS are only those issued before 952 seconds. When I/O latencies are longer than 18.65 ms, the CDF value of FastScale is greater than that of SLAS. This indicates again that FastScale has a smaller maximum response time for user I/Os than SLAS. The average latency of FastScale is close to that of SLAS: 8.01 ms and 7.53 ms respectively. It is noteworthy that due to significantly shorter data redistribution time, FastScale has a markedly smaller impact on user I/O latencies than does SLAS.

A factor that might affect the benefits of FastScale is the workload under which data redistribution performs. Under the TPC-C workload, we also measure the performance of FastScale and SLAS to perform the “4+2” scaling operation.

For the TPC-C workload, Figure 23 shows local maximum latencies versus the redistribution times for SLAS and FastScale. It once again shows the efficiency of FastScale in improving the redistribution time. The redistribution times using SLAS and FastScale are 6820 seconds and 964 seconds, respectively. That is to say, FastScale causes an improvement of 85.87% in the redistribution time. Likewise, local maximum latencies of FastScale are also obviously shorter than those of SLAS. The global maximum latency using FastScale is 114.76 ms while that using SLAS reaches 147.82 ms.

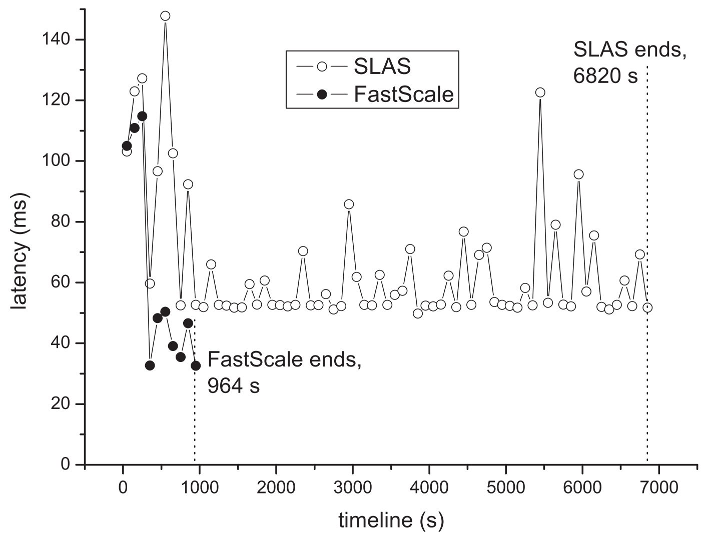



Fig. 23. Performance comparison between FastScale and SLAS under the TPC-C workload.

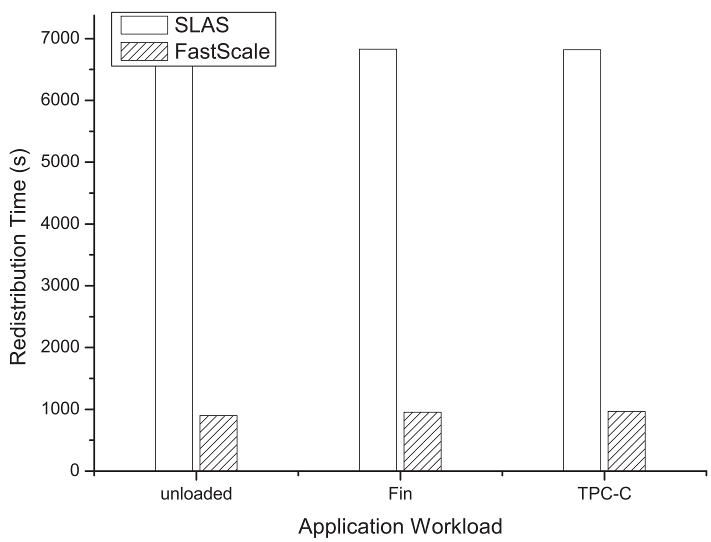



Fig. 24. Comparison of redistribution times of FastScale and SLAS under different workloads. The label "unloaded" means scaling a RAID volume offline.

To compare the performance of FastScale under different workloads, Figure 24 shows a comparison in the redistribution time between FastScale and SLAS. For completeness, we also conducted a comparison experiment on the redistribution time with no loaded workload. To scale a RAID volume offline, SLAS uses 6802 seconds whereas FastScale consumes only 901 seconds. FastScale provides an improvement of 86.75% in the redistribution time.

We can draw one conclusion from Figure 24. Under various workloads, FastScale consistently outperforms SLAS by 85.87–86.75% in the redistribution time, with shorter maximum response time for user I/Os.

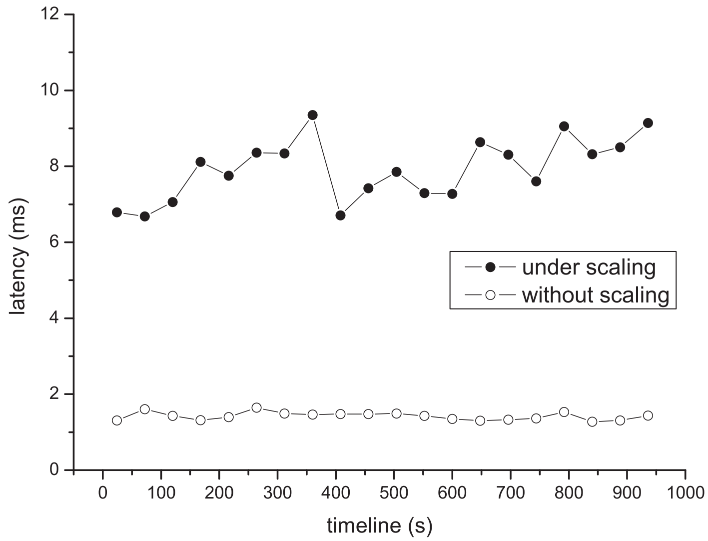



Fig. 25. Impact of RAID-0 scaling on application I/O performance under the Fin workload.

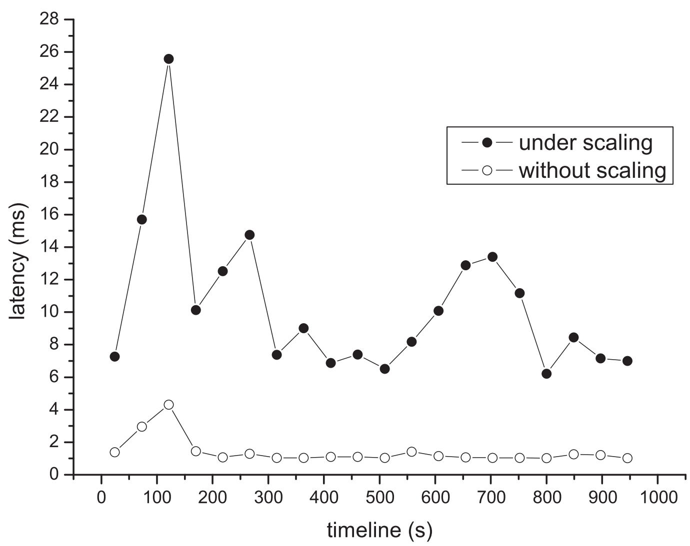



Fig. 26. Impact of RAID-0 scaling on application I/O performance under the TPC-C workload.

To quantitatively demonstrate how data migration affects the existing workload during RAID scaling using FastScale, we measure the performance of RAID-0 without scaling operations. The measured array is made up of four disks. Figure 25 plots local average latencies with and without RAID scaling under the Fin workload, as the time increases along the $x$-axis. The average latency without scaling is 1.43 ms, while the average latency during RAID scaling is 8.01 ms. Figure 26 plots local average latencies with and without RAID scaling under the TPC-C workload. The average latency without scaling is 1.38 ms, while the average latency during RAID scaling is 10.39 ms.

Obviously, application I/O latencies during RAID scaling are higher than those without scaling. The reason behind this phenomenon is that application I/Os are continuous

Table II. Comparison of the Improvement of Redistribution Times with Different Disk Sizes

| Disk Size | 4 GB | 8 GB | 16 GB |
|---|---|---|---|
| SLAS | $6,802s$ | $13,603s$ | $27,206s$ |
| FastScale | $901s$ | $1,801s$ | $3,598s$ |
| Improvement Percentage | 86.75% | 86.76% | 86.77% |

without long idle periods, either under the Fin workload or under the TPC-C workload. Interleaving of redistribution I/Os will increase the time of application I/Os waiting for processing, and the time of disk seeks. The rate-control parameter can be used to trade off between the redistribution time objective and the response time objective. In order to obtain acceptable application I/O latencies, one can adjust the rate-control parameter—the IOPS threshold. In other words, FastScale can accelerate RAID-0 scaling, and at the same time, have an acceptable impact on the existing workload.

Running a simulation experiment is time consuming. We set the disk capacity to 4 GB for the online scaling, so as to conduct an experiment in an acceptable time. We also perform some experiments of offline scaling, where the disk capacity is set 8 GB and 16 GB. As shown in Table II, the redistribution time increases linearly with the disk size used, no matter which approach is used. However, the percentages of improvement are consistent with those with the 4 GB capacity.

*4.3.2. The Performance after Scaling.* The preceding experiments show that FastScale improves the scaling efficiency of RAID significantly. One of our concerns is whether there is a penalty in the performance of the data layout after scaling using FastScale, compared with the round-robin layout preserved by SLAS.

We use the Web workload to measure the performance of the two RAIDs, scaled from the same RAID using SLAS and FastScale. Each experiment lasts 500 seconds, and records the latency of each I/O. Based on the issue time, the I/O latency sequence is evenly divided into 20 sections. Furthermore, we get a local average latency from each section.

First, we compare the performance of the two RAIDs, after one scaling operation “4+1” using the two scaling approaches. Figure 27 plots local average latencies for the two RAIDs as the time increases along the $x$-axis. We find that the performance of the two RAIDs are very close. With regard to the round-robin RAID, the average latency is 11.36 ms. For the FastScale RAID, the average latency is 11.37 ms.

Second, we compare the performance of the two RAIDs, after two scaling operations “4+1+1” using the two approaches. Figure 28 plots local average latencies of the two RAIDs as the time increases along the $x$-axis. It again reveals approximate equality in the performance of the two RAIDs. With regard to the round-robin RAID, the average latency is 11.21 ms. For the FastScale RAID, the average latency is 11.03 ms.

Third, we compare the performance of the two RAIDs, after three scaling operations “4+1+1+1” using the two approaches. Figure 29 plots local average latencies of the two RAIDs as time increases along the $x$-axis. It again reveals the approximate equality in the performance of the two RAIDs. With regard to the round-robin RAID, the average latency is 11.01 ms. For the FastScale RAID, the average latency is 10.79 ms.

Finally, we compare the performance of the two RAIDs, after four scaling operations “4+1+1+1+1” using the two approaches. Figure 30 plots local average latencies of the two RAIDs as time increases along the $x$-axis. It again reveals the approximate equality in the performance of the two RAIDs. With regard to the round-robin RAID, the average latency is 10.75 ms. For the FastScale RAID, the average latency is 10.63 ms.

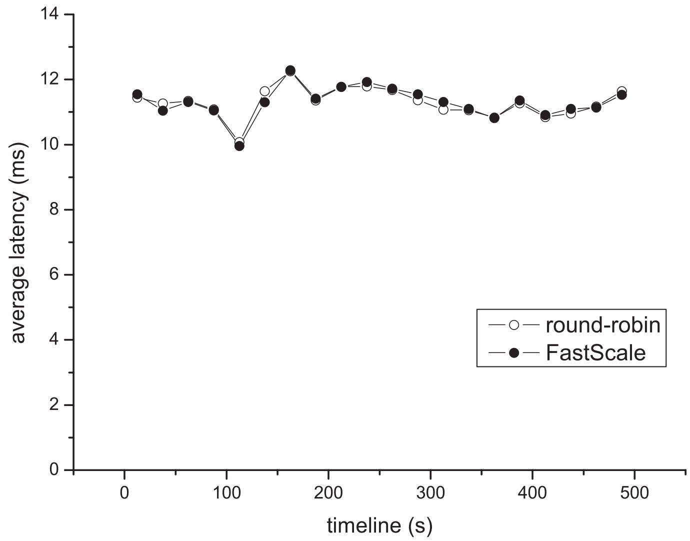



Fig. 27. Performance comparison between FastScale's layout and round-robin layout under the Web workload after one scaling operation, i.e., "4+1".

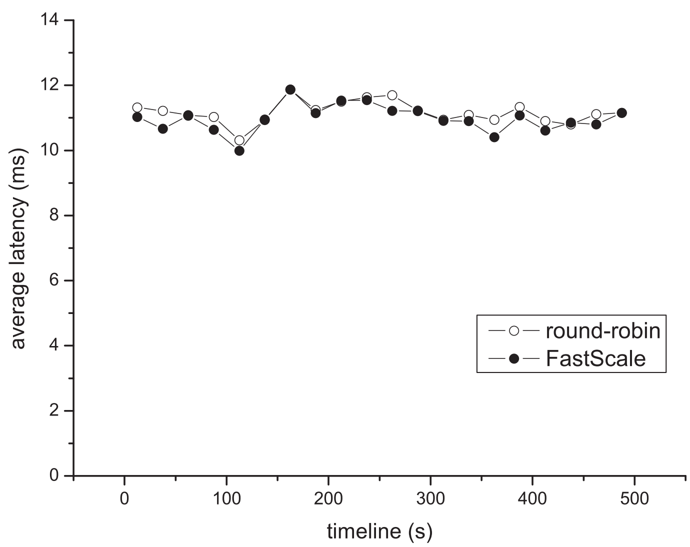



Fig. 28. Performance comparison between FastScale's layout and round-robin layout under the Web workload after two scaling operations, i.e., "4+1+1".

To summarize, Figure 31 shows a comparison in the response times of the two RAIDs, scaled from the same RAID using SLAS and FastScale, as the number of scaling times increases along the $x$-axis. We can see that the response time of each RAID decreases as scaling times increase. This is due to an increase of the number of disks in a RAID, which serve user I/Os simultaneously. The other conclusion that can be reached is that the two RAIDs, scaled from the same RAID using SLAS and FastScale, have almost identical performance. In three of the four cases, the FastScale RAID even performs better than the round-robin RAID.

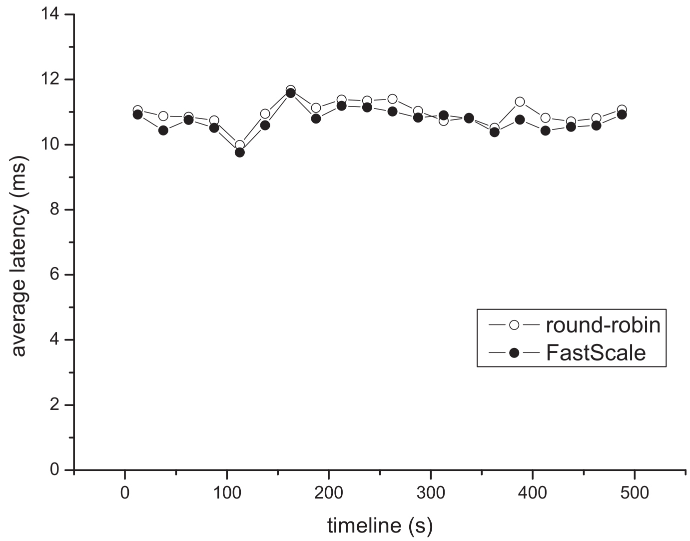



Fig. 29. Performance comparison between FastScale's layout and round-robin layout under the Web workload after three scaling operations, i.e., "4+1+1+1".

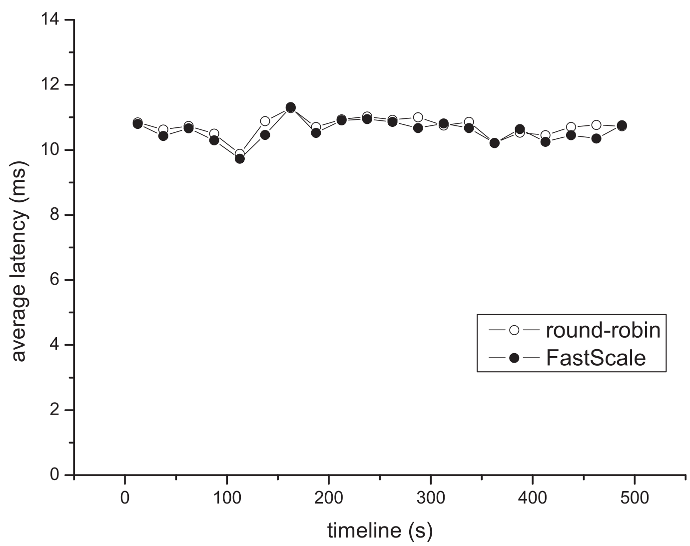



Fig. 30. Performance comparison between FastScale's layout and round-robin layout under the Web workload after four scaling operations, i.e., "4+1+1+1+1".

## 5. RELATED WORK

In this section, we first examine the existing approaches to scaling deterministic RAID. Then, we analyze some approaches to scaling randomized RAID.

### 5.1. Scaling Deterministic RAID

HP AutoRAID [Wilkes et al. 1996] allows an online capacity expansion. Newly created RAID-5 volumes use all of the disks in the system, but previously created RAID-5 volumes continue to use only the original disks. This expansion does not require data

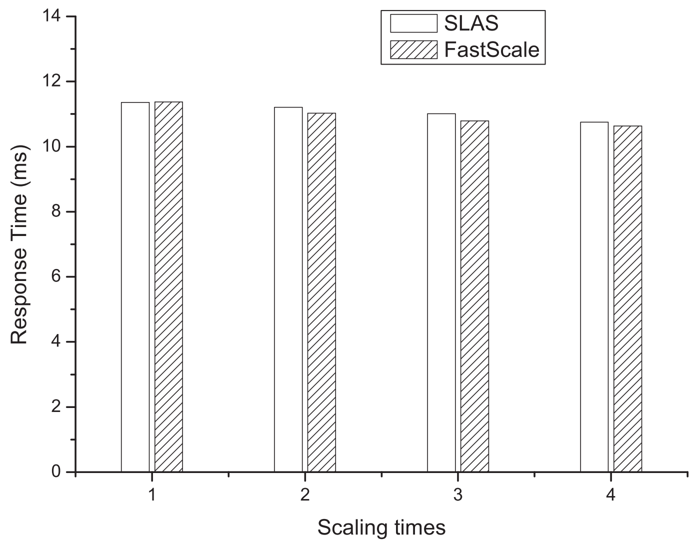



Fig. 31. Comparison of response times of the two RAIDs, scaled from the same RAID using SLAS and FastScale.

migration. However, the system cannot add new disks into an existing RAID-5 volume. The conventional approaches to RAID scaling redistribute data and preserve the round-robin order after adding disks.

Gonzalez and Cortes [2004] proposed a gradual assimilation algorithm (GA) to control the overhead of scaling a RAID-5 volume. However, GA accesses only one block via an I/O. Moreover, it writes mapping metadata onto disks immediately after redistributing each stripe. As a result, GA has a large redistribution cost.

The reshape toolkit in the Linux MD driver (MD-Reshape) [Brown 2006] writes mapping metadata for each fixed-sized data window. However, user requests to the data window have to queue up until all data blocks within the window are moved. On the other hand, MD-Reshape issues very small (4KB) I/O operations for data redistribution. This limits the redistribution performance due to more disk seeks.

Zhang et al. [2007] discovered that there is always a reordering window during data redistribution for round-robin RAID scaling. The data inside the reordering window can migrate in any order without overwriting any valid data. By leveraging this insight, they proposed the SLAS approach, improving the efficiency of data redistribution. However, SLAS still requires migrating all data. Therefore, RAID scaling remains costly.

D-GRAID [Sivathanu et al. 2004] restores only live file system data to a hot spare so as to recover from failures quickly. Likewise, it can accelerate the redistribution process if only the live data blocks from the perspective of file systems are redistributed. However, this requires semantically-smart storage systems. On the contrary, FastScale is independent of file systems, and it can work with any ordinary disk storage.

A patent [Legg 1999] presented a method to eliminate the need to rewrite the original data blocks and parity blocks on original disks. However, the method makes all the parity blocks be either only on original disks or only on new disks. The obvious distribution nonuniformity of parity blocks will bring a penalty to write performance.

Franklin and Wong [2006] presented a RAID scaling method using spare space with immediate access to new space. First, old data are distributed among the set of data disk drives and at least one new disk drive while, at the same time, new data are mapped to the spare space. Upon completion of the distribution, new data are copied from the spare space to the set of data disk drives. This is similar to the key idea of WorkOut [Wu et al. 2009]. This kind of method requires spare disks to be available in the RAID.

In another patent, Hetzler [2008] presented a method of RAID-5 scaling, called MDM. MDM exchanges some data blocks between original disks and new disks. MDM can perform RAID scaling with reduced data movement. However, it does not increase (just maintains) the data storage efficiency after scaling. The RAID scaling process exploited by FastScale is favored in this regard, because the data storage efficiency is maximized, which many practitioners consider desirable.

AdaptiveZ [Gonzalez and Cortes 2007] divides the space of RAID into several adaptive zones, whose stripe patterns can be customized separately. When new disks are added, AdaptiveZ adds a new zone and redistributes a part of the data at the end of the RAID in the zone. This results in more blocks allocated on old disks than new ones, while redistributing the minimum amount of blocks that the AdaptiveZ algorithm requires. Therefore, AdaptiveZ has to increase the size of the migrated zone. In other words, AdaptiveZ is faced with a dilemma between minimal data migration and even data distribution. On the contrary, FastScale combines minimal data migration and uniform data distribution. On the other hand, all the data migrated by AdaptiveZ are logically sequential. On account of spatial locality in I/O workloads, this will still comprise a balanced load.

### 5.2. Scaling Randomized RAID

Randomized RAID [Alemany and Thathachar 1997; Brinkmann et al. 2000; Goel et al. 2002; Santos et al. 2000] appears to have better scalability. It is now gaining the spotlight in the data placement area. Brinkmann et al. [2000] proposed the cut-and-paste placement strategy that uses randomized allocation strategy to place data across disks. For a disk addition, it cuts off the range $[1/(n+1), 1/n]$ from given $n$ disks, and pastes them to the newly added $(n+1)$th disk. For a disk removal, it uses a reversing operation to move all the blocks in disks that will be removed to the other disks. Also based on random data placement, Seo and Zimmermann [2005] proposed an approach to finding a sequence of disk additions and removals for the disk replacement problem. The goal is to minimize the data migration cost. Both of these approaches assume the existence of a high-quality hash function that assigns all the data blocks in the system to uniformly distributed real numbers with high probability. However, they did not present such a hash function.

The SCADDAR algorithm [Goel et al. 2002] uses a pseudo-random function to distribute data blocks randomly across all disks. It keeps track of the locations of data blocks after multiple disk reorganizations and minimizes the amount of data to be moved. Unfortunately, the pseudo-hash function does not preserve the randomness of the data layout after several disk additions or deletions [Seo and Zimmermann 2005]. So far, a truly randomized hash function that preserves its randomness after several disk additions or deletions has not been found.

The simulation report in Alemany and Thathachar [1997] shows that a single copy of data in random striping may result in some hiccups of the continuous display. To address this issue, one can use data replication [Santos et al. 2000], where a fraction of the data blocks randomly selected are replicated on randomly selected disks. However, this will incur a large overhead.

RUSH [Honicky and Miller 2003, 2004] and CRUSH [Weil et al. 2006] are two algorithms for online placement and reorganization of replicated data. They are probabilistically optimal in distributing data evenly and minimizing data movement when new storage is added to the system. There are three differences between them and

FastScale. First, they depend on the existence of a high-quality random function, which is difficult to generate. Second, they are designed for object-based storage systems. They focus on how a data object is mapped to a disk, without considering the data layout of each individual disk. Third, our mapping function needs to be 1-1 and on, but hash functions have collisions and count on some amount of sparseness.

## 6. CONCLUSIONS AND FUTURE WORK

This article presents the design of a new approach called FastScale, which accelerates RAID-0 scaling by minimizing data migration. First, with a new and elastic addressing function, FastScale minimizes the number of data blocks to be migrated without compromising the uniformity of data distribution. Second, FastScale uses access aggregation and lazy checkpoint to optimize data migration.

Replaying real-system disk I/O traces, we evaluated the performance of FastScale through comparison with an efficient scaling approach called SLAS. The results from detailed experiments show that FastScale can reduce redistribution time by up to 86.06% with smaller maximum response time of user I/Os. The experiments also illustrate that the performance of the RAID scaled using FastScale is almost identical to, or even better than, that of round-robin RAID.

In this article, the factor of data parity is not taken into account. We believe that FastScale provides a good starting point for efficient scaling of RAID-4 and RAID-5 arrays. In the future, we will focus on the scaling issues of RAID-4 and RAID-5.

## ACKNOWLEDGMENTS

We would like to thank the three anonymous reviewers for their constructive comments which have helped to improve the quality and presentation of this article.

## REFERENCES

Alemany, J. and Thathachar, J. S. 1997. Random striping news on demand servers. Tech. rep. TR-97-02-02, University of Washington.

Brigham Young University. 2010. TPC-C Postgres 20 iterations. DTB v1.1. Performance Evaluation Laboratory, Trace distribution center. http://tds.cs.byu.edu/tds/.

Brinkmann, A., Salzwedel, K., and Scheideler, C. 2000. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*. 119–128.

Brown, N. 2006. Online RAID-5 resizing. drivers/md/ raid5.c in the source code of Linux Kernel 2.6.18. http://www.kernel.org/.

Bucy, J., Schindler, J., Schlosser, S., and Ganger, G. 2008. The DiskSim Simulation Environment Version 4.0 Reference Manual. Tech. rep. CMU-PDL-08-101, Carnegie Mellon University.

Franklin, C. R. and Wong, J. T. 2006. Expansion of RAID subsystems using spare space with immediate access to new space. US Patent 10/033,997.

Goel, A., Shahabi, C., Yao, S., and Zimmermann, R. 2002. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*. 473–482.

Gonzalez, J. L. and Cortes, T. 2004. Increasing the capacity of RAID5 by online gradual assimilation. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*. 17–24.

Gonzalez, J. L. and Cortes, T. 2007. Adaptive data block placement based on deterministic zones (AdaptiveZ). In *Lecture Notes in Computer Science*, vol. 4804, 1214–1232.

Hennessy, J. and Patterson, D. 2003. *Computer Architecture: A Quantitative Approach*, 3rd Ed. Morgan Kaufmann Publishers, Inc., San Francisco, CA.

Hetzler, S. R. 2008. Data storage array scaling method and system with minimal data movement. US Patent 20080276057.

Hitachi. 2001. Hard disk drive specifications Ultrastar 36Z15. http://www.hitachigst.com/tech/techlib.nsf/techdocs/85256AB8006A31E587256A7800739FEB/$file/U36Z15 sp10.PDF. Revision 1.0, April.

Honicky, R. J. and Miller, E. L. 2003. A fast algorithm for online placement and reorganization of replicated data. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*.

Honicky, R. J. and Miller, E. L. 2004. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*.

Kim, C., Kim, G., and Shin, B. 2001. Volume management in SAN environment. In *Proceedings of the 8th International Conference on Parallel and Distributed Systems (ICPADS)*. 500–505.

Legg, C. B. 1999. Method of increasing the storage capacity of a level five RAID disk array by adding, in a single step, a new parity block and N-1 new data blocks which respectively reside in new columns, where N is at least two. US Patent: 6000010, December 1999.

Muller, K. and Vignaux, T. 2009. SimPy 2.0.1 documentation. http://simpy.sourceforge.net/SimPyDocs/index.html.

Patterson, D. A. 2002. A simple way to estimate the cost of down-time. In *Proceedings of the 16th Large Installation Systems Administration Conference (LISA)*. 185–188.

Patterson, D. A., Gibson, G. A., and Katz, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the International Conference on Management of Date (SIGMOD)*. 109–116.

Santos, J. R., Muntz, R. R., and Ribeiro-Neto, B. A. 2000. Comparing random data allocation and data striping in multimedia servers. *ACM SIGMETRICS Perform. Eval. Rev. 28*, 1, 44–55.

Seo, B. and Zimmermann, R. 2005. Efficient disk replacement and data migration algorithms for large disk subsystems. *ACM Trans. Storage 1*, 3, 316–345.

Sivathanu, M., Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2004. Improving storage system availability with D-GRAID. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*.

Storage Performance Council. 2010. http://www.storageperformance.org/home.

UMass Trace Repository. 2007. OLTP Application I/O and Search Engine I/O. http://traces.cs.umass.edu/index.php/Storage/Storage.

Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. 2006. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the International Conference on Supercomputing (SC)*.

Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. 1996. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1, 108–136.

Wu, S. J., Jiang, H., Feng, D., Tian, L., and Mao, B. 2009. WorkOut: I/O workload outsourcing for boosting the RAID reconstruction performance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*. 239–252.

Zhang, G. Y., Shu, J. W., Xue, W., and Zheng, W. M. 2007. SLAS: An efficient approach to scaling round-robin striped volumes. *ACM Trans. Storage 3*, 1, 1–39.

Zheng, W. M. and Zhang, G. Y. 2011. FastScale: Accelerate RAID scaling by minimizing data migration. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*.

Received March 2012; revised August 2012; accepted May 2013