TRAID: Exploiting Temporal Redundancy and Spatial Redundancy to Boost Transaction Processing Systems Performance

Pengju Shang, Student Member, IEEE, Saba Sehrish, and Jun Wang, Senior Member, IEEE

Abstract—In the past few years, more storage system applications have employed transaction processing techniques to ensure data integrity and consistency. Logging is one of the key requirements to ensure transaction Atomicity, Consistency, Isolation, Durability (ACID) properties and data recoverability in transaction processing systems (TPS). Recently, emerging complex I/O bound transactions have resulted in substantially more log content and higher log flushing latency. The latency will delay transaction commit and decrease the overall throughput of the TPS. On the other hand, RAID is widely used as the underlying storage system for Databases to guarantee system reliability and availability with high I/O performance. In this paper, we observe the overlap between the redundancies in the underlying RAID storage system and database logging system, and propose a novel reliable storage architecture called Transactional RAID (TRAID). TRAID deduplicates this overlap by only logging one compact version (XOR results) of recovery references for the updating data. It minimizes the amount of log content and thereby boosts the overall transaction processing performance. At the same time, TRAID guarantees the same RAID reliability, as well as recovery correctness and ACID semantics as current TPS setups. We experiment on two open-source database systems: Berkeley DB and PostgreSQL, with three different workloads: standard OLTP benchmark TPC-C, customized TPC-C with strong access locality, and customized TPC-C with write-intensive property. Then we test TRAID performance with “Group Commit” enabled. Finally, we evaluate the recovery efficiency of TRAID. Our extensive results demonstrate that for throughput, TRAID outperforms RAID by 43.24-69.5 percent for various workloads; it also saves on log space by 28.57-35.48 percent, and outperforms RAID by about 20 percent in throughput with “Group Commit” enabled. At last, we show that TRAID outperforms RAID from 28.7 to 35.7 percent during the recovery.

Index Terms—RAID, transaction processing system, log, ACID.

1 INTRODUCTION

Logging is the key mechanism to guarantee the durability and correctness of transactions [1], [2], [3], [4] and has been playing an increasingly important role in Transaction Processing Systems (TPS). Recent years have seen increasing number of I/O bound transaction processing applications, for example, in Temporal databases [5] and Multidimensional databases [6], [7]. The log latency increases significantly in these systems because: 1) more object activities need to be logged and 2) the description of a single object includes more multidimensional data sources. The latency caused by logging denotes the wait time before a transaction commits: the locks on the updating data cannot be released and the transaction cannot commit until the log is flushed onto stable storage devices. This longer log latency results in committing a fewer number of transactions in a particular time frame and is becoming the bottleneck of the overall transaction processing performance.

DRAM capacity doubles every two years [8], so a normal OLTP database that was considered “large” can now fit into main memory easily. However, TPS always requires the log data to write to stable media before the transaction can commit. This protocol is known as write-ahead-logging (WAL). In other words, log latency is mainly determined by the amount of log data which is going to be flushed and the I/O bandwidth of the log devices. Meanwhile, with the increasing popularity of flash memory and flash disks, several works [9], [10], [11], [12] have been proposed to use these new types of storage devices for database logging. Flash devices do not suffer from the small sequential log writes (9X faster than magnetic disks) since they have no moving components. At the same time, their capacities have been increasing and the price per GB decreasing exponentially [13]. However, flash devices are still expensive at $5-10 per GB in 2009 (compared to magnetic disks at $0.1-0.2 per GB), and the existing research do not solve the increasing log size issue. The solution we proposed in this paper could be a complement to flash device-based logging systems by decreasing almost half of the log size and therefore saving the budget on the needed capacity.

In this paper, we develop a new Transactional RAID (TRAID) to attack the long log latency problem for transaction processing applications. The idea is to deduplicate the information redundancy at different layers, e.g., temporal redundancy (i.e., different versions of data copies in the time domain) on the database’s log disk and spatial redundancy (i.e., parity redundancy or mirroring redundancy) in the...
RAID architecture. Our design is based on the observation that highly reliable RAID systems are widely used in commercial databases [14], [15], [16], such as RAID5 and RAId10. For these database systems, there exists an overlap between databases log disk and underlying storage system: the spatial redundancy provided by the disk arrays is often overlooked by the database and file system designers. On the other hand, the disk array designers are often unaware of the fault tolerant mechanisms deployed by the upper level file systems and database management systems. As a result, both groups tend to implement an independent fault tolerant system from their own perspectives and thereby leading to potentially high overhead.

TRAID exploits this overlap to improve the overall performance without violating the ACID [17], [18] properties and recovery correctness of transactions. The databases using erasure coded disk arrays allow us to deduplicate redundancy by exploiting an extra XOR result of old data and new data. The advantage of this method is simplicity: undo operation sets the page to the old value, while redo sets it to the new one; both undo and redo are idempotent, which is not so trivial in other logging methods. The disadvantage is large log space and long log latency: one operation, such as splitting a B-tree, may result in tens or hundreds of physical log records. In order to solve the problems of Physical Logging, Logical Logging [18] is developed. Logical Logging records the name of an undo-redo operation and its parameters, rather than the values. This method achieves the smallest logging space and the shortest latency; however, it may fail in media recovery. Moreover, it assumes that each action is atomic. This assumption is often violated when the action is partially complete and the system cannot decide how far to undo. It is a big problem in transaction processing where the undo operation may not be idempotent. As a result, Physiological Logging [18] is developed, which uses logical logging when possible and uses physical logging when higher reliability is required. This method transforms one complex action into a sequence of messages and page actions. These are structured as minitransactions and log records are generated for the changes in pages or session state. In other words, Physiological Logging is a fine-grain logging method, where both the old and new values for the updating page or record are logged when necessary.

Besides the logging schemes, tremendous works have been done to further improve the logging efficiency. In commercial databases, e.g., SQL Server 2008, techniques like regular backup truncates old log records no longer needed for recovery to prevent the log files from consuming all of the disk space [3]. Log compression [20], [21] allows log records to be compressed and decompressed as they are written and read from the log files, which can provide disk usage savings. Bulk-logged [22] option in SQL Server reduces the penalty of logging because the following operations are minimally logged and not fully recoverable: SELECT INTO, bulk-load operations, CREATE INDEX as well as text and image operations. Hence, any-point-in-time recovery is not possible with bulk-logged option. The group commit option [19] accumulates several parity updates into one bigger parity update by employing journaling techniques, so that the “write penalty” for small writes can be alleviated. However, all of these logging methods and adjustments are based on the trade-off between performance and correctness (ACID properties). Our solution in this paper exploits a new perspective to improve logging efficiency without violating ACID by utilizing the data redundancy in the storage systems.

3 TRAID DESIGN

TRAID is implemented as a reliable RAID storage for TPS, and reduces log latency by minimizing log size. As a result, besides performance and reliability, we also show how redo/undo operations are performed correctly, i.e., recovery correctness and also the ACID semantics provided by relational database systems are maintained in TRAID.

TRAID design exploits the overlap between the existing spatial redundancy in the most commonly used RAID architectures, e.g., parity based (RAID5) redundancy or mirroring-based (RAID10) and temporal redundancy in log disks.

3.1 Parity Redundancy: TRAID5

RAID5 is a representative storage system with parity redundancy. In database systems on RAID5, besides the original data block, there are two more copies of the same block in the system. Upon a block update request, both the before and after image of the updating block are saved to the log disk. Hence, one copy (with two versions) is on
the log disk, while the other copy can be generated on the fly by using the RAID5 parity and the other blocks on the same stripe. The overlap between these two copies at any point in time enables us to log-less data. More specifically, we log the XOR result of the old parity and new parity instead of before and after images of the updating block. This XOR result can provide enough references for undo and redo operations when it works with the copy in spatial redundancy. In this way, the overlap between the spatial (parity) redundancy in RAIDs and the temporal redundancy in database log is eliminated. The XOR of successive update operations when it works with the copy in spatial redundancy.

Before going to the details of TRAID5 design, we recall how RAID5 processes a block update transaction.

1. Reading the target block and the parity on the same stripe from disk to memory.
2. Calculating the new parity.
3. Writing the raw data and updated data into log file for undo and redo operations. Transaction can commit after Step 3.
4. Writing the updated data and new parity onto disk.

The parity $P$ in RAID5 is calculated as follows: suppose at time $T_1$, we have $(A_1, B_1, C_1, P_1)$ in RAID5, where $P_1 = A_1 \oplus B_1 \oplus C_1$; at time $T_2$, one update request changes $A_1$ to $A_2$, then $P_2 = A_2 \oplus A_1 \oplus P_1 = A_2 \oplus B_1 \oplus C_1$.

In TRAID5 design, we add one new XOR result defined as TRAID-parity. Instead of logging old and new block data in Step (3), we log: $Q = P_1 \oplus P_2 = A_1 \oplus A_2$. The comparison of TRAID5 architecture with RAID5 is shown in Fig. 1.

The update transaction is processed in TRAID5 in 3 steps as follow:

1. Reading the block and corresponding parity information from disk into memory.
2. Calculating the new RAID parity $P$ and TRAID-parity $Q$, writing $Q$ and all other transaction information into the log file; the transaction can commit after Step (2); no physical undo or redo data are required, and
3. Writing the updated block and parity $P$ onto disk.

In this way, the log space is decreased. Because of WAL protocol, less log content in TRAID5 decreases the log flushing time, as well as the transaction committing time.

Although, the above TRAID-parity equation is for single-block-update cases, it can be adopted for multi-block-update cases. As we know, the write penalty in RAID5 (the extra disk I/O to calculate the new parity) is alleviated by combining the “small writes” into one “big write.” However in memory, the calculation of the new parity still requires several steps to cover all the updating blocks. For example, one update request on the stripe containing $(A_1, B_1, C_1, P_1)$ in RAID5 may want to get a result like $(A_2, B_2, C_1, P_2)$. The one-time write of $P_2$ can be $P_2 = A_2 \oplus B_2 \oplus C_1$. But before that, we will have two versions of $P$ in the memory, such as $P_2 = A_2 \oplus A_1 \oplus P_1 = A_2 \oplus B_1 \oplus C_1$ and $P_2 = B_2 \oplus B_1 \oplus P_2 = A_2 \oplus B_2 \oplus C_1$, where $P_2$ equals to $P_1$. TRAID-parity of block “A” and block “B” are obtained by taking advantage of these intermediate calculations: $Q_A = P_1 \oplus P_2 = A_2 \oplus A_1$ and $Q_B = P_2 \oplus P_1 = B_2 \oplus B_1$. Hence, TRAID5 would not bring any extra I/O while still guaranteeing one $P$ write as RAID5 does. For simplicity, in the following discussion about the TRAID-parity calculation, we only consider a single-block-update case.

Above equations are for the transactions updating the block only once. Given a transaction which will update the block multiple times during its lifetime, we need to calculate a $Q$ list recording all the updates. However, for these multiple-update transactions, the way to calculate $Q$ will depend on whether complete rollback or partial rollback is required. We discuss these two cases in the following two sections.

### 3.1.1 Complete Rollback

A complete rollback means that we need to reset the database to the original state when undo is needed. For example, one transaction updates block $A$ for $K$ times from time $T_1$ to $T_K$. Complete rollback will require the system to be reset to the status at $T_1$, rather than other intermediate statuses. In this case, we only record the newest TRAID-parity info (at time of point $T_K$) $Q_K$ for the updates on block $A$ as follows: $Q_1 = \phi$ when $T = 1$, where $\phi$ denotes $NULL$; $Q_2 = P_2 \oplus P_1 \oplus Q_1$.
TABLE 1
TRAID-Parity Calculation for Complete Transaction Rollback in TRAID5

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
<th>Parity P</th>
<th>Parity Q</th>
<th>Get A</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(0)</td>
<td>Initialize</td>
<td>P₀ = A ⊕ B ⊕ C</td>
<td>Q₀ = NULL</td>
<td>A = A</td>
</tr>
<tr>
<td>T(1)</td>
<td>A → A₁</td>
<td>P₁ = A₁ ⊕ A ⊕ P₀</td>
<td>Q₁ = P₁ ⊕ P₀</td>
<td>A₁ = A₁ ⊕ Q₁</td>
</tr>
<tr>
<td>T(2)</td>
<td>A₁ → A₂</td>
<td>P₂ = A₂ ⊕ A₁ ⊕ P₁</td>
<td>Q₂ = P₂ ⊕ P₁ ⊕ Q₁</td>
<td>A₂ = A₂ ⊕ Q₂</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>T(K)</td>
<td>Aₖ₋₁ → Aₖ</td>
<td>Pₖ = Aₖ ⊕ Aₖ₋₁ ⊕ Pₖ₋₁</td>
<td>Qₖ = Pₖ ⊕ Pₖ₋₁ ⊕ Qₖ₋₁</td>
<td>A = Aₖ ⊕ Qₖ</td>
</tr>
</tbody>
</table>

when T = 2; Qₖ = Pₖ ⊕ Pₖ₋₁ ⊕ Qₖ₋₁ when K ≥ 2. If the old data A₁ are lost, Qₖ guarantees that old data can be recovered by A₁ = Qₖ ⊕ Aₖ. Similarly, if the new data Aₖ are lost, the XOR result of Qₖ and old data A₁ obtains the new data (redo to Aₖ). Table 1 shows the details of recovery in case of a complete rollback to A₁.

3.1.2 Partial Rollback
In real database environment, a transaction supporting partial rollback can write to the disk several times before it commits. In this case, we need a list of Q parities, i.e., Q₁, Q₂, ..., Qₙ, for all the writes as some or all of them will be used for the partial rollback. Q for the partial rollback is calculated as follows: Q₁ = P₁ ⊕ P₀ when T = 1, Q₂ = Pₙ ⊕ P₁ when T = 2, and Qₖ = Pₖ ⊕ Pₖ₋₁ when T ≥ 2. If there is a system failure at a time point m with data Aₘ and the database needs to rollback to Aₘ at time point of m, where 1 ≤ m < n, undo operation to Aₘ will work as follows: Aₘ = Aₘ ⊕ Qₘ ⊕ Qₘ₋₁ ⊕ ... ⊕ Qₙ₊₁. Having Q₁, Q₂, ..., Qₙ and A₁, we also can redo this transaction to any point of time m, where 1 ≤ m ≤ n, by the following calculation: Aₘ = A₁ ⊕ Q₂ ⊕ ... ⊕ Qₙ. The details of partial recovery of data is shown in Table 2.

It may be noted that the TRAID5 technique can be easily ported to build TRAID6 and other erasure coded arrays. The double-parity RAID or parity-based RAID6—such as RDP [25]—maintains two parities and parity is used to calculate the parity Q.

3.1.3 Recovery Correctness
TRAID5 guarantees the recovery correctness after failures. In Section 3.1, we list the three steps which TRAID5 needs to process an “update” transaction. If the system fails after step 2, the database is still in a consistent state and no recovery is needed.

If there is a system failure after step 2 and before step 3, we may have two different cases: 1) the transaction is committed before the failure, redo is needed for the recovery. The XOR result of Q in the log and the unupdated data on the disk are used for redo, and the parity P can be calculated again and 2) the transaction is not committed yet, since data on disk are not updated, we can just mark this transaction as failed.

If there is a system failure during step 3, then it means the write operation is interrupted and the block is neither old nor updated (partially corrupted). The RAID5 parity information and checksum guarantee that we can undo this failed update to get the original data (before update). Then, there are two cases: 1) the transaction is already committed, we can redo the transaction by using the XOR result of RAID-parity and the original data and 2) the transaction is not committed yet, the original data we get from the RAID5 parity are the unupdated version, no further undo is needed.

If the transaction needs to be aborted after step 3 (the real data have been updated but the transaction has not committed yet; this case can be caused by limited data buffer, or user-defined synchronization), TRAID-parity and the updated data can be used to undo this write to the original data.

TABLE 2
TRAID-Parity Calculation for Partial Transaction Rollback in TRAID5

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
<th>Parity P</th>
<th>Parity Q</th>
<th>Get any version of A</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(0)</td>
<td>Initialize</td>
<td>P₀ = A ⊕ B ⊕ C</td>
<td>Q₀ = NULL</td>
<td>A = A</td>
</tr>
<tr>
<td>T(1)</td>
<td>A → A₁</td>
<td>P₁ = A₁ ⊕ A ⊕ P₀</td>
<td>Q₁ = P₁ ⊕ P₀</td>
<td>A₁ = A₁ ⊕ Q₁</td>
</tr>
<tr>
<td>T(2)</td>
<td>A₁ → A₂</td>
<td>P₂ = A₂ ⊕ A₁ ⊕ P₁</td>
<td>Q₂ = P₂ ⊕ P₁</td>
<td>A₂ = A₂ ⊕ Q₂</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>T(K)</td>
<td>Aₖ₋₁ → Aₖ</td>
<td>Pₖ = Aₖ ⊕ Aₖ₋₁ ⊕ Pₖ₋₁</td>
<td>Qₖ = Pₖ ⊕ Pₖ₋₁</td>
<td>Aₖ = Aₖ ⊕ Qₖ ⊕ ... ⊕ Qᵢ₊₁, 0 ≤ i &lt; K</td>
</tr>
</tbody>
</table>
the database modifications follow the “all or nothing” rule. As a result, ACID properties are well maintained in DB + TRAID5 systems.

It may be noted that the TRAID5 technique can be easily ported to build TRAID6 and other erasure coded arrays. The double-parity RAID or parity-based RAID6—such as RDP [25]—maintains two parities $P$ and $P'$. $P$ is same as the RAID5 parity and $P'$ is used for the recovery of second disk failure. The spatial recovery requirement is different for RAID5 and RAID6, but the temporal recovery (undo/redo on a particular drive at time domain) provided by TRAID-parity $Q$ is the same. Hence, only $P$ parity is used to calculate the TRAID-parity $Q$.

3.2 Mirroring Redundancy: TRAID10

RAID10 combines mirroring redundancy (RAID1) and striping (RAID0). Every striped block in the RAID0 part has a mirroring block in its RAID1 partner. We exploit the overlap between the temporal redundancy in the log disk and spatial redundancy in the mirroring copies to implement TRAID10.

A database running on RAID10 processes an update transaction in following steps:

1. Read the requested data from disks into the memory.
2. Write the Before Image (e.g., $A_1$) into the log for undo requests.
3. Write the After Image (e.g., $A'_1$) into the log for redo requests. Transaction cannot commit until the log is flushed due to WAL protocol.
4. Update the data anytime before or after transaction commits.

We add an XOR operation to TRAID10. Instead of logging the old data and new data, TRAID10 also records their XOR result in the log file. As shown in Fig. 2, an update request in TRAID10 is processed as follows:

1. Read the requested data into memory.
2. Calculate the TRAID-parity $Q$ based on both old version and new version data; write $Q$ into the log file; transaction can commit after Step (2).
3. Update the data anytime before or after transaction commits.

Since the mirroring redundancy from the underlying RAID10 will provide ideal reliability for either old data (before the data on disk are updated) or new data, we will always have the reference for transaction undo and redo. In this way, we utilize the overlap between the temporal redundancy and spatial redundancy to reduce log space and log flushing latency, thereby accelerate the transaction commit procedure.

The way of calculating $Q$ in TRAID10 also depends on whether complete rollback or partial rollback is adopted, as shown in the following two sections.

3.2.1 Complete Rollback

We use the same example in TRAID5: one transaction updates block $A$ for $K$ times from time $T_i$ to $T_K$, and needs to completely rollback from the current $A_i$ where $1 \leq i \leq K$, to the original $A$ at time $T_0$. We calculate the TRAID-parity $Q$ as Table 3 shows.

The “Get $A$” column in Table 3 shows how to do the complete rollback from time $T_i$ where $1 \leq i \leq K$ to $T_0$. For the committed transactions, if system fails before the data on disks get updated (we have the original version of $A$), TRAID-parity $Q$ can be used to get $A_K$ by calculating $A_K = A \oplus Q_K$. Formally, $Q$ is able to calculate $A_K$ from any intermediate version of $A$, denoted as $A_i$, where $0 \leq i < K$, by calculating $A_K = A_i \oplus Q_K$.

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
<th>TRAID-parity $Q$</th>
<th>Get $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(0)$</td>
<td>Initialize</td>
<td>$Q_0 = NULL$</td>
<td>$A = A_0$</td>
</tr>
<tr>
<td>$T(1)$</td>
<td>$A \rightarrow A_1$</td>
<td>$Q_1 = A \oplus A_1$</td>
<td>$A = A_1 \oplus Q_1$</td>
</tr>
<tr>
<td>$T(2)$</td>
<td>$A_1 \rightarrow A_2$</td>
<td>$Q_2 = A_1 \oplus A_2 \oplus Q_1$</td>
<td>$A = A_2 \oplus Q_2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$T(K)$</td>
<td>$A_{K-1} \rightarrow A_K$</td>
<td>$Q_K = A_{K-1} \oplus A_K \oplus Q_{K-1}$</td>
<td>$A = A_K \oplus Q_K$</td>
</tr>
</tbody>
</table>
3.2.2 Partial Rollback
We need a list of \( Q \) parities in TRAID10 for partial rollback. Using the same example as above, we need to record every \( Q_i \) at time \( T_i \), where \( 1 \leq i \leq K \). The detail calculation of \( Q \) list \( Q_1, Q_2, \ldots, Q_K \) is shown in Table 4.

The “Get any version of A” column in Table 4 shows how to use \( Q \) and the current version of \( A \) on disk to roll the transaction backward to any point of time. If the system fails after the transaction committed but before the data are updated to the final version \( A_K \), we can get \( A_K \) from the current \( A_i \) where \( 0 \leq i < K \) by calculating \( A_K = A_i \oplus Q_{i+1} \oplus Q_{i+2} \oplus \cdots \oplus Q_K \).

3.2.3 Recovery Correctness
TRAID10 guarantees the recovery correctness after failures. If the system fails after or during step 2 before the transaction commits, neither undo nor redo is necessary since no change is made to the disk and the transaction is not committed.

Assuming an updating transaction is committed after step 2, a system failure may require the transaction to redo. In this case, the TRAID-parity \( Q \) or \( Q \) list in the log will be used as redo reference. Also, note that the transaction is already committed, we must not undo because of Durability property in ACID. If a transaction failed at step 3 before it committed, we can use the calculations shown in Table 3 and 4 to roll the transaction backward to any point of time for future redo or the original version as undo.

3.2.4 ACID Semantics
Similar to TRAID5, TRAID10 can also guarantee ACID semantics of Transaction processing as traditional storage systems do. We exclude the analysis due to the space limitation.

3.3 Other Design Issues

Data version check. We use TRAID-Parity \( Q \) with the updated data for both TRAID5 and TRAID10 to perform undo, and with the old data to do redo. In order to implement such recovery function, we need to know whether the data on disk are old or new. We resort to the existing checksum [26], [27], [28] solutions for data version check. Parity-based RAID systems always turn to checksum to detect data corruption. A checksum is a fixed-size datum computed from hash functions, fingerprints, and so forth. When the data block is being updated, a checksum is calculated based on a function \( f \) of the new data. RAID can use the checksum to tell whether the update is complete or has failed. TRAID can use this checksum to do a data version check: upon a recover request, we use the same \( f \) based on the current data on disk. If this result is the same as the checksum calculated before, it means the update operation has finished successfully, and the data on disk are new; otherwise, the data on disk are the old version.

Group commit. TRAID can easily work with other log I/O optimization methods, such as group commit [19], to gain incremental performance benefit. In a database with group commit, instead of starting a WAL flush immediately after a commit record is inserted, it waits for a while to give other backends a chance to finish their transactions and have them flushed by one log I/O. There are two parameters related to group commit: how many commits to wait for (commit group size), and how often (timeout) to flush the dirty log buffer. Given a specified commit group size, since TRAID can shrink a single log record size by avoiding the before image for update and replace transactions, one log I/O in TRAID can commit even more transactions.

Adopting group commit will require the system to maintain the commit ordering and the serialization of the transactions. All the related requirements just affect the lock table and transaction table [29], but have nothing to do with the log content. On the other hand, TRAID optimizes the log I/O by exploiting the overlap between the spatial and temporal redundancy so that the log content of every update transaction is reduced. As a result, TRAID complements other log I/O optimization methods, such as group commit, etc.

Log space. The aforementioned discussion is based on the assumption that log is recorded in either an inexpensive disk or disk arrays, where disk I/O is expensive. If log is stored on some expensive storage devices, such as NVRAM or flash memory, where the storage space is the main concern, TRAID can reduce the log size to improve the efficiency of space utilization.

Locking granularity. Almost all database systems support fine granularity locking besides page-level locking, such as record-level locking (for high concurrency requirements), object-level locking (for object-oriented database systems), etc. [17]. For example, in Berkeley DB, page-level locking is adopted for creating the database file, since a page is the basic allocation unit. While record-level locking may be enabled for updates to records that are on the page, in order to achieve high concurrency. The data structures of undo and redo information in the log are affected by the locking level. The whole page (record) content is logged for page-level (record-level) locking, if needed [30]. However, different granularity locking does not affect the recovery process. For example, in ARIES [1], both page-level and record-level lockings are supported in a uniform fashion in the log subsystem. In other words, no matter whether the whole page or only the record, or even only the changed attribute is logged, the recovery works in the same way: undo based on the original image, redo based on the new.

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
<th>Parity ( Q )</th>
<th>Get any version of ( A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(0)</td>
<td>Initialize</td>
<td>( Q_0 = NULL )</td>
<td>( A = A_0 )</td>
</tr>
<tr>
<td>T(1)</td>
<td>( A \rightarrow A_1 )</td>
<td>( Q_1 = A_1 \oplus A_0 )</td>
<td>( A = A_1 \oplus Q_1 )</td>
</tr>
<tr>
<td>T(2)</td>
<td>( A_1 \rightarrow A_2 )</td>
<td>( Q_2 = A_2 \oplus A_1 )</td>
<td>( A = A_2 \oplus Q_2 \oplus Q_1; A_1 = A_2 \oplus Q_2 )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>T(K)</td>
<td>( A_{K-1} \rightarrow A_K )</td>
<td>( Q_K = A_K \oplus A_{K-1} )</td>
<td>( A_i = A_K \oplus Q_K \oplus \cdots \oplus Q_{i+1} ) for ( 0 \leq i &lt; K )</td>
</tr>
</tbody>
</table>
image. Our design avoids the redundant original image (no matter which level locking is being used) stored in the log file when a database is working with RAID architecture.

**Modularity loss.** TRAID utilizes the overlapped redundancy between database log space and RAID storage space, hence these two levels are unified at some extent which will result in some loss in modularity. In TRAID, the new generated log which contains the TRAID-parity will depend on the setting of underlying RAID system (strip size, block size, number of disks, and so on). As a result, the modularity between the storage system and application level is potentially compromised. In the future work, we want to implement an abstraction layer to eliminate this trade-off.

**Data reliability of TRAID.** RAID architecture was developed to enhance the reliability of the multiskin subsystem. TRAID also wants to keep the same data reliability while boost the performance of transaction processing.

The only difference between databases using TRAID and RAID is the log content, which does not affect the reliability of the storage system. For RAID5 and TRAID5, if one disk failed, the failed block can be recovered by one XOR calculation on the available data and parity information; more than one disk failure will result in date loss since there is no enough redundancy information to do the recovery. TRAID-Parity Q can only be used for undo or redo according to the transaction requirement. In other words, recording Q instead of old and new data will only result in different recovery mechanisms in time domain but has nothing to do with storage system reliability. As a result, the data reliability of RAID5 and TRAID5 is same.

Let \( N \) be the number of disks in the TRAID5 and RAID5, \( MTTF_{disk} \) be the mean time to failure for each disk, \( MTTR \) be the mean repair time. Hence, the \( MTTDL \) (mean time to data loss) of TRAID5 and RAID5 are given by:

\[
MTTDL_{TRAID5} = MTTDL_{RAID5} = \frac{MTTF_{disk}}{(N-1) \cdot C^2} \cdot MTTR
\]

Similarly, TRAID10 and RAID10 have the same reliability because both of them will not lose data unless the two disks belonging to the same mirroring group failed. The \( MTTDL \) of TRAID10 and RAID10 are given by:

\[
MTTDL_{TRAID10} = MTTDL_{RAID10} = \frac{2 \cdot MTTF_{disk}^2}{N \times MTTR}
\]

### 4 Experimental Setup

#### 4.1 Testbed

In order to implement TRAID5 and TRAID10, we modify the corresponding RAID codes in Linux kernel version 2.6.11 on a Dell Precision 690 (Intel Xeon E5345—2.33 GHz/4.0 GB RAM). Five uniform 250G SATA disk drives with 7,200 rpm rotation speed are installed. The benchmarks are TPC and tailored TPC(s). We create soft (T)RAID5 (with four disks—three data disks and one parity disk, another disk was used as log disk) and (T)RAID10 (with four disks, the last disk is the log disk). We launch the experiments on top of two open-source database systems: Berkeley DB 4.3 version [30] and PostgreSQL (PGS) 8.1.4 version [31]. There is another well-known open-source database system MySQL, which uses similar logging method as Berkeley DB (Rollback Segments). We chose to hack Berkeley DB because it is more light-weight and simple. PostgreSQL is considered because it uses a different transaction logging scheme, named Multi-version Concurrency Control (MVCC). Moreover, Berkeley DB uses page-level logging while PostgreSQL uses record-level logging by default. We will evaluate our TRAID on top of these two different logging systems.

#### 4.2 Implementation of TRAID5 and TRAID10

For simplicity, we only consider the transactions without partial commit, so TRAID5 and TRAID10 just need one version of TRAID_Parity.

In TRAID5, we add one TRAID5-parity calculation step using the existing XOR engine in RAID5. A hook is also added to the XOR_block function in the RAID5 source code to get the required block information and write the calculated TRAID-Parity into the buffer. When the buffer is full, or the size of group commit limitation is reached or the database decides to write the updated transaction data to the disk, the TRAID-parities are flushed to the log disk.

In TRAID10, we create an XOR-calculation function because there is no XOR engine. One hook is added in the RAID10 controller to record two versions of the block which is to be updated. The old version and new version of the data are the inputs for XOR calculation.

#### 4.3 Workloads

In order to have a fair evaluation of TRAID, we use three benchmarks: a commercial benchmark for transaction processing evaluation: TPC-C [32], and two modified versions of TPC-C as microbenchmarks.

The first benchmark, TPC-C, simulates an Online Transaction Processing (OLTP) database environment. It can measure the performance of a system which is tasked with processing numerous short business transactions concurrently. It is set in the context of a wholesale supplier operating on a number of warehouses and their associated sales districts. TPC-C incorporates five types of transactions with different complexity for online and deferred execution on a database system. These transactions perform the basic operations on databases such as inserts, deletes, updates, and so on. The transactions in TPC-C and their percentage of the transaction mix are [33]:

1. New Order (45 percent): read-write.
2. Payment (43 percent): read-write.
3. Order Status (4 percent): read-only.
4. Stock Level (4 percent): read-only.
5. Delivery (4 percent): read-write.

Based on the implementation of standard TPC-C, we developed a special version of TPC-C for our test, named BTPC-C1 (biased TPC-C benchmark1). In BTPC-C1, the key values in the queries and updates were changed from a uniformly random distribution to a biased distribution in the form of 90/10 rules. In this way, we increase the access locality so that the resulting workload is more sensitive to lock content delay, and the lock-log content delay. By using BTPC-C1, one locked transaction can cause more transactions to wait for the lock release, so we can see how much benefit can be gained by using TRAID, i.e., a reduced log-lock latency. In the experiments with BTPC-C1, we increase the number of concurrent processes to see the performance of DB + TRAID and DB + RAID systems.
The third benchmark aims to test the performance of TRAID with a write-intensive workload, called BTPC-C2 (biased TPC-C benchmark). In BTPC-C2, we shield all the read-only transactions in TPC-C. Because read requests in TRAID and RAID are identical, read-intensive transactions may obviate the performance improvement. Therefore, by using BTPC-C2, we can explore the advantages of TRAID for the transactions with dominant update requests.

It is necessary to consider the locking-level issue: it is known that there are page-level locking and record-level locking in database. We want to show the performance improvement of TRAID is not limited to any specific locking level. Hence, the block size of a TRAID-parity is set to 512 Bytes, which is same as the default page size in Berkeley DB and PostgreSQL. In our experiments, the page size is 512 Bytes, the record size in WAREHOUSE Table is about 480 Bytes, in CUSTOMER Table is over 700 Bytes, and in STOCK Table is about 420 Bytes; while in other five tables, the record sizes range from 100 to 200 Bytes. Ninety-two percent of the transactions will read/write the first two tables, the record sizes range from 100 to 200 Bytes. Ninety-two percent of the transactions will read/write the first three tables. For the majority of time, only one record can be fit in one page, which means the page-level locking in our experimental configuration is comparable to record-level locking.

5 Experimental Results

We run the TPC-C benchmark workload with the warehouse parameter set to 20, representing an initialized database size of 4 GB, which grows during each test run as new records are inserted. In our TPC-C benchmark, the input includes number of transactions, number of terminals (number of concurrent processes). The output consists of the transaction processing time and the transactions per minute (tpmC). For each test, we run the given number of transactions ten times and get the average response time to analyze the TRAID performance in addition to the size of log file in each experiment. Berkeley DB and PostgreSQL are denoted as BDB and PGS for short, respectively.

5.1 Experiments on BDB

In Berkeley DB, the logging method is similar as the Rollback Segments-based schemes in Oracle, MySQL [34]. These database systems simply store the modified data of each transaction in the data tables; meanwhile, the information required to rollback any particular transaction is stored in the rollback segments (in log files). BDB uses page-level logging.

5.1.1 Standard TPC-C Benchmark

The first experiment compares the overall response time of BDB + RAID and BDB + TRAID for a given number of transactions. In standard TPC-C, we set the number of concurrent processes to 10 (as the official setting), and the number of warm-up transactions to 1,000. Fig. 3 shows the overall execution times of TPC-C on RAID10, RAID5, TRAID10, and TRAID5; Fig. 4 shows the corresponding throughputs.

From Fig. 3, we can see that compared to RAID10 and RAID5, TRAID10 and TRAID5 improves the overall response time significantly, and the improvement increases with the increasing number of transactions. The average throughput of RAID10 in Fig. 4 is 230.21 tpmC, while the average throughput of TRAID10 is 329.71 tpmC, which means BDB+TRAID10 is 43.23 percent faster than BDB+RAID10. Similar conclusions are drawn for RAID5 and TRAID5, the average throughput of RAID5 is 197.4 tpmC, while the one of TRAID5 is 310.5 tpmC, which means TRAID5 outperforms RAID5 by 56.89 percent.

There are two reasons behind this improvement: 1) instead of writing two versions of the updating page or record into the log disk, TRAID only writes two-thirds of the amount of data, which will decrease the log flushing time as well as the time when the updating transactions are locked and 2) in order to maximize the I/O bandwidth utilization, database systems usually buffer the updating writes and wait until the buffer is full to execute a big I/O. For the logging system, a similar mechanism is also used via a log buffer (group commit). Since we decreased the log content significantly, the buffer with same size for group commit in TRAID systems can hold more committing transactions compared to RAID systems, hence one same log flush I/O operation in TRAID can commit more transactions. The latter one is the key to realize the significant improvement of overall response time.

The improvement of TRAID5 over RAID5 is more significant than that of TRAID10 over RAID10, because the TRAID5-parity is the intermediate result of RAID5 parity calculation. For example, upon one request to update page $A$ to $A'$, the RAID5 parity calculation needs to do $P' = P \oplus A \oplus A'$ while TRAID5-parity is part of it: $Q = P \oplus P' = A \oplus A'$. As a result, we can simply read and record $Q$ from the $P'$ calculation rather than doing one extra calculation. However, in TRAID10, we need to implement a real XOR function and do a real calculation for TRAID10-parity $Q$.

The throughput of TRAID5/RAID5 is a little less than TRAID10/RAID10 because we do not implement any extra optimization to eliminate the write penalty in RAID5 or TRAID5, while TPC-C has a larger percentage of writes than reads.
TRAID is also evaluated for log size improvement as shown in Fig. 5. The size of log files in BDB+RAID10 and BDB+RAID5 is the same since all the pages being updated (including the before and after images) are logged in Berkeley DB. Both BDB + TRAID10 and BDB + TRAID5 only record the TRAID-parity information instead of before and after images. From Fig. 5, we can see that TRAID10 saves the log space up to 33.7 percent compared to a RAID system, while TRAID5 saves 32.6 percent.

Before analyzing this result, note that we cannot avoid recording the regular transaction information in the log file: 1) LSN, Transaction ID, etc., 2) some other logged operations, such as page allocation, keep track of record counts in a B tree, mark a record on a page as deleted, etc., and 3) the relative large checkpoint records in BDB log file, which log all the pages are being accessed by the running transactions. Hence, by only recording the TRAID-parity in TRAID, the log size is reduced by one-third rather than 50 percent.

As we mentioned above, we set the parity block size as 512 Bytes in TRAID and it is the basic unit for parity computations. Actual data sizes of disk write requests (stripe size) are independent of the parity block size but are aligned with parity blocks. With this setting, one TRAID-parity will take as much space as one Before_image does in BDB log file, and it is the only way to make a fair size comparison of the TRAID5 log with Berkeley DB log. As a result, theoretically the log size of TRAID5 should be similar to TRAID10. The small difference in the experiment result is due to the different response time: TRAID5 needs a little bit more time to do all the transactions, which may result in several more checkpoint records (a new checkpoint is made every 60 seconds).

5.1.2 BTPC-C1 with Access Locality

The second experiment with BTPC-C1 benchmark evaluates the impact of data access locality on the TRAID performance. Since in BTPC-C1 90 percent of the queries and update requests focus on 10 percent of the data, the overall performance will be more sensitive to the log-lock-latency effect. With the increasing number of concurrent processes, the benefit of TRAID over RAID becomes more significant because TRAID reduces the wait time of subsequent transactions. We run 10,000 transactions implemented by BTPC-C1, and gradually increase the number of concurrent processes. The overall corresponding throughputs are shown in Fig. 6.

From Fig. 6, we can see the performance improvement from RAID to TRAID is not substantial when there is only one process. The difference between TRAID and RAID in this case (sequentially transaction processing) is the waiting-time of log-writing for sequential transactions. Also, since no concurrent transactions exist, there is no log-locking time which can further delay the transaction commit time.

The trend of throughput improvement for different number of concurrent processes is shown in the Fig. 7. It is clear that the throughput improvement from RAID to TRAID increases gradually with the number of concurrent transactions up till five, and then the improvement factor starts decreasing. The lock-content delay is a crucial factor in transaction response time before the number of concurrent processes reaches five. TRAID gains more improvement with the increasing concurrency and more lock contention because it can decrease the log-lock content delay. However, after this point, the disk I/O costs dominate the transaction response time while the lock content effect has reached the peak. The throughput improvement of TRAID over RAID becomes stable (between 41.9 to 49.6 percent) after the concurrency reaches a threshold of five, where the improvement is maximized as 79.6 percent for TRAID10 and 63.7 percent for TRAID5.

5.1.3 Write-Intensive BTPC-C2

The third experiment tests the performance of TRAID for write-intensive workloads, BTPC-C2, in which every transaction needs to read and update the database. We changed the percentages of five transaction mix by deleting...
the read-only transactions such as Order Status transactions and Stock Level transactions, and increasing the percentages of the other three kinds of transactions. In this experiment, we set the number of concurrent processes to five, which is the turning point of performance improvement according to Section 5.1.2. The overall execution times of BDB + RAID10, BDB + RAID5, BDB + TRAID10, and BDB + TRAID5 are shown in Fig. 8.

By calculating the average improvement, TRAID10 outperforms RAID10 by 69.5 percent while TRAID5 outperforms RAID5 by 62.7 percent. Recall these numbers with standard TPC-C benchmark in the first experiment, the TRAID10 and TRAID5 outperform RAID10 and RAID5 by 43.23 and 56.89 percent, respectively. Hence, TRAID obtains more improvement for write-intensive workloads because more updates of the transactions are needed to be logged. Meanwhile, more write requests result in higher possibility of resource conflicts, which force more transactions to wait for the conflicting transaction commit. This log-locking time for a transaction commit can be reduced by TRAID. Therefore, TRAID performs even better for write-intensive workloads.

5.2 Experiments on PostgreSQL
PostgreSQL uses MVCC-based logging method, which is different from Berkeley DB. Rows of a table in PGS are stored as tuples; two fields of each tuple are xmin and xmax, which record the transaction ID of the transaction creating and deleting this tuple, respectively. Insertions in PGS will generate one tuple (in both database table and log files) with the xmax blank and the xmin set to the transaction ID. Deletions in PGS will find the tuple in database table and set the xmax field; but in sequencing log files, PGS will add a new deleting log record rather than looking up and modifying the existing tuple because log buffer operations are much faster than accessing the log files on disk. Update in PGS is no more than a concurrent insert and delete, which will write both the new tuple and the old tuple into the log files.1 PGS uses record-level logging.

5.2.1 Log Size and Latency
In order to highlight the log size and logging latency, we use PGS to generate the database according to the TPC-C criterions. The database generating process includes insertions, deletions, and updates, all of which will write log files. The results about log size are shown in Fig. 9. In PGS, each log file is 16M by default. When it is full, a new log file is generated; there are at most seven log files at one time; thereafter, the oldest one will be removed as the new one is added.

Fig. 9 shows that the real data size of warehouses’s tables is relatively small when compared with the log size. On average, log size is 6.01-6.74 times as large as the size of generated database. When using TRAID, the log size reduces by 28.57-35.48 percent for TRAID10 and 25.37-31.03 percent for TRAID5.

For the logging latency, it should be noted that PGS has two functions to perform WAL logging: XLogInsert and XLogFlush. XLogInsert is used to place a new record into the WAL buffers during the transaction processing phase; XLogFlush is made at transaction commit time to ensure that transaction records are flushed to permanent storage. We add a timer in XLogFlush function to quantify the log flushing latencies, as illustrated in Fig. 10.

The results show that the logging latencies on regular RAID devices are 81.9-82.08 percent of the overall time. When using TRAID, the above numbers become 54.9-60.5 percent for TRAID10 and 58.9-62.2 percent for TRAID5. On average, TRAID can reduce the log latency by 26.7-30.6 percent. Although these latencies are related to the I/O speed of the storage devices, we only focus on the improvement ratio by using TRAID.
5.2.2 Throughput of TRAID

We use a open-source implementation of TPC-C, named TPCC-UVA [35] on PostgreSQL engine to evaluate the performance of TRAID. We run the test with different numbers of warehouses and fixed 10 terminals in each warehouse. The ramp-up period is set to 20 minutes and the measurement period is set to two hours. The results are shown in Fig. 11.

Based on the results, TRAID10 outperforms RAID10 by 54.3 percent, TRAID5 outperforms RAID5 by 57.5 percent on average. It is interesting to observe that TRAID only reduces the log latency by 26.7-30.6 percent, while it can improve the overall throughput over 50 percent. The reason is the conflict transactions need to wait less in TRAID; a small reduction in log latency may help more transactions to proceed or commit earlier. As a result, TRAID can comparably improve the transaction processing efficiency for both page-level logging (BDB) and record-level logging (PGS).

5.3 TRAID and Group Commit

Group commit can be used in some databases to improve the log I/O efficiency. In this set of experiments, we show that TRAID combined with group commit can further improve transaction processing throughput. The database we use is Berkeley DB.

We enable group commit in BDB by setting `DB_TXN_WRITE_NOSYNC` to the database environment, which is disabled by default. If `DB_TXN_WRITE_NOSYNC` is set, Berkeley DB will write, but not synchronously flush the log on transaction commit. This means that transactions do not exhibit the durability requirement of the ACID semantics. Database integrity will be maintained. However, if the application or system fails, some number of the most recently committed transactions may be undone during recovery [36]. The number of transactions at risk is governed by how many log updates can fit into the log buffer, how often the operating system flushes dirty buffers to disk (controlled by a data structure called `Timer`), and how often the log is checkpointed. We disable the checkpoint in Berkeley DB and do not set the Timer so that the log buffer will not flush to disks unless it is full. In other words, the log buffer size is exactly the same as group commit size.

Before doing the experiment, we need to give an optimized range of log buffer size in a busy, high-DML (Data Manipulation Language) database. Common wait events related to a too-small log buffer size include high “redo log space requests” and a too-large log buffer may result in high “log file sync” waits. Burleson [37] recommends that a value of 10 MB for the log buffer is a reasonable value for Oracle Applications and it represents a balance between concurrent programs and online users, and the value of log buffer must be a multiple of log buffer block size, 512 bytes. Following these rules, 1M to 10M log buffer sizes are used in the experiment. After setting the log buffer size, we run 10,000 TPC-C transactions on top of BDB + RAID with group_commit (GC for short) and BDB + TRAID with GC. The first 200 and the last 200 transactions are used for warm up and cool down which are not counted in the overall performance. The results are shown in Fig. 13.

We list all the throughput improvement percentages of BDB + TRAID with GC over BDB + RAID with GC in Table 5. TRAID5 + GC outperforms RAID5 + GC by 17.8 percent on average, while TRAID10 + GC outperforms RAID10 + GC by 18.9 percent on average. It is interesting to observe that the throughput improvement is decreasing along with the increasing group commit size. With a larger log buffer size, more transactions to commit can fit in the buffered group; the log I/O efficiency is being improved. But the improvement is decreasing because the impact of log I/O cost is getting smaller. When log I/O latency is no longer the main bottleneck, we cannot improve the overall performance significantly by only adopting group commit. As a result, the improvement of transaction throughput comes down to about 11 percent and stabilizes afterward. The reason is that one log buffer flush (group commit) in TRAID can commit more transactions.

5.4 Rollback Performance

TRAID and RAID use different ways to undo transactions, so it is interesting to compare their respective rollback performance. We create transactions modifying some values of the records in CUSTOMER table, such as the customer’s name, address, etc. All the transactions will be processed to the end but rolled back before it is committed. Only one process is generated to run this simulation because we just want to highlight the rollback performance. We run 500 to 3,000 transactions sequentially and record the whole execution time including the transaction processing time plus the rollback time. The results are shown in Fig. 14, the corresponding improvements are shown in Fig. 15.

Since the transactions in this section are write intensive, we compare Fig. 14 with Fig. 12. We first notice that RAID10 always outperforms RAID5 in nonrollback situation in
Fig. 12, but it is not always true when the transactions need to be rolled back, as shown in Fig. 14. When the transactions are processing normally, RAID10 is doing better due to the write penalty of RAID5; while during the transaction rollback, RAID5 could do it with higher parallelism. The read from log disk will collect all the data that the undoing transaction needs, RAID5 in our experiment (consists of four disks) could undo the transaction on three disks (not including the one parity disk) at the same time, while RAID10 does it only on two mirroring groups. Our second finding is that although the overall performance of TRAID10/5 is always better than that of RAID10/5 no matter whether there is rolled back or not, the improvement is decreased when we include the rolled back transactions.

Fig. 15 shows that when the transactions are rolled back, the average improvement of TRAID10 is 35.7 percent, while TRAID5 is 28.6 percent. These numbers in Section 5.1.3 are 47.4 and 61.7 percent, respectively. The reason is that our new rollback methods result in more disk I/O. For example, the undo operation on single block in RAID5 only results in one read (from log disk), one write (to the block on data disk); while in TRAID5, we will have one read (from log disk), one read (from data disk), one XOR calculation, and one write (to the block on data disk), plus the extra I/O to update the parity information on the same stripe. But this overhead is still trivial when compared with the benefit TRAID brought. The most interesting find is that when we include rollback, the overall performance of TRAID5 is decreased more than that of TRAID10 (33.1 percent compared with 11.7 percent). This is still caused by the write penalty in RAID5. When the transaction is rolled back, we write the calculated value (the old value) to the disk, this write will in turn cause extra I/O to calculate the new RAID5 parity; while in TRAID10, we introduce less overhead (one read from data disk, one XOR calculation, and one write) since there is no parity and related write penalty.

6 CONCLUSIONS

In this paper, we have presented a Transactional RAID (TRAID) design for transaction processing applications. TRAID exploits the existing information redundancies in RAID and database systems to minimize the log size. We have implemented TRAID5 and TRAID10 systems for erasure coded disk array and replica-based disk array, respectively. TRAID only logs the TRAID-parity which provides enough recovery references. TRAID5 uses the old and new parity redundancies to calculate the new TRAID-parity, whereas TRAID10 directly uses the old and new version of updating data. We also show that TRAID offers recovery correctness, ACID semantics, and reliability comparable to Database on RAID. TRAID also works with other log I/O optimization such as group commit to obtain even better transaction throughput. Our results demonstrate that TRAID performs similarly for both page-level logging and record-level logging: for throughput, TRAID outperforms RAID by 43.24-69.5 percent for various workloads; it also saves on log space by 28.57-35.48 percent, and outperforms RAID by about 20 percent in throughput when “Group Commit” is enabled. Finally, we show that TRAID outperforms RAID from 28.7 to 35.7 percent during the recovery.

REFERENCES

Pengju Shang received the BS degree from Jilin University, Changchun, China, the MS degree from Huazhong University of Science and Technology, Wuhan, China. Currently, he is working toward the PhD degree in computer engineering school of University of Central Florida, Orlando. His specific interests include transaction processing, RAID systems, storage architecture, power management in storage systems, CMP architectures, Hadoop/MapReduce architecture, and cloud computing. He is a student member of the IEEE.

Saba Sehrish received the PhD degree under the supervision of Dr. Jun Wang from the Department of Electrical Engineering and Computer Science, University of Central Florida in 2010. She is a postdoctoral researcher at Northwestern University, working with Professor Alok Choudhry. Her research interests include parallel and distributed I/O Systems, high performance, data-intensive and cloud computing.

Jun Wang received the PhD degree in computer science and engineering from the University of Cincinnati in 2002. He is an associate professor and Charles N. Milican faculty fellow in EECS and the director of the Computer Architecture and Storage Systems (CASS) Laboratory at the University of Central Florida, Orlando, Florida. He has authored more than 60 publications in premier journals such as IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, and leading HPC and systems conferences such as IPDPS, HPDC, EuroSys, ICS, Middleware, FAST. He has conducted extensive research in the areas of Computer Systems and High Performance Computing. His specific research interests include massive storage and file system in local, distributed and parallel systems environment. He is the recipient of a US National Science Foundation Early Career Award 2009 and Department of Energy Early Career Principal Investigator Award 2005. He has graduated six PhD students who upon their graduations were employed by major US IT corporations (e.g., Google, Microsoft, etc.). He has served as numerous US National Science Foundation (NSF) grant principal investigators and US Department of Energy (DOE) grant principal investigators and TPC members for many premier conferences. He has cochaired the First International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED 2008) held together with HPCA. He is a senior member of the IEEE and the IEEE Computer Society.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.