Decoupled Address Updates

Abstract

Reliable storage systems often create a copy of data object on an update request to prevent system from running into a data inconsistent state. Depending on how to make the new object visible to the clients, there are two mechanisms used in existing file systems: object update and address update. For example, journaling in file systems is an object update mechanism while copy on write (COW) is an address update, more specifically a coupled address update mechanism. These two mechanisms often lead to multiple writes and recursive updates issues. In this paper, we identify that the addressing function used in both mechanisms can be enhanced to overcome the limitations. We introduce a new mechanism called decoupled address updates that adopts an one-to-two/many addressing function, which maps one logical object ID (e.g., file name) to several physical addresses. This enables the storage system to manage both data and its copies in a unified space in a more efficient way. By experimenting with file systems benchmarks on our prototype systems, we conclude that 1-n addressing function obtains much better performance as compared to 1-1 addressing functions in Ext3 and ReiserFS for many representative cases.

1 Introduction

A storage system typically adopts different kinds of metadata and system data to manage application data in an efficient fashion. The best known examples are perhaps file systems and database systems. Our increasing dependence on storage systems has not only enhanced our standard of living, but has also left us vulnerable to ever increasing errors and accidental faults. These hardware errors and faults damage the data consistency in storage systems and can potentially be introduced in every step in developing and utilizing storage systems [1, 2, 3].

Moreover, recent years have seen the addition of many new functionalities to storage systems such as enabling versioning service [4, 5, 6], dynamic metadata structures [7], range queries [8], and file system transaction semantics [9, 10, 11, 12, 13]. Also, in the file systems, a myriad of metadata infrastructures have been generated by emerging data-intensive applications, such as email clients (*e.g.*, sendmail), different kinds of search engines [14] (*e.g.*, desktop, enterprise, metasearch, index, *etc*), multimedia management applications, *etc*. These infrastructures expand the horizon of traditional file systems consistency semantics among metadata, to consistency between data and data, among multiple file operations, multiple files, and even different file systems. In short, system failures and the trends in storage systems have imposed new challenges to revisit the existing consistency solutions.

In file systems, consistency means that systems enforce pre-defined integrity constraints among storage objects (*inc.* both data and metadata). Existing consistency schemes often replicate the essential object at different addresses to guard against data damage and tend to impact the performance negatively especially when large data objects are involved. To access a storage object, clients query the *addressing function* with a logical address (*e.g.*, file block number or database page number). The addressing function is a common mechanism to translate the logical address to a physical address (*e.g.*, physical disk block number—PBN) in the non-volatile storage system space. Upon an update request, the system writes the new copy to a different location before it persistently modifies a storage object. Otherwise the system could enter an inconsistent state if a crash takes place between the start and completion of the update. Depending on how to make the new object visible to clients, we classify the existing mechanisms into two categories [1]: *object update* which overwrites the original object, and *address update* which changes both the logical and physical address of the object (*i.e.*, value of object's address). More specifically, we call this address update mechanism in the existing solutions a *coupled address update*. Journaling in file systems [17, 18] is an object update mechanism, while copy-on-write (COW) [19, 4] is a coupled address update method.

Despite being successfully practiced for several decades, these mechanisms have several major limitations.

- Object update method writes new objects at different addresses, and then overwrites the old objects. New objects are normally written in a log fashion to improve performance, but also require crash recovery. Coupled address update mechanism writes new objects one by one and no crash recovery is needed. The ever increasing amount of data and metadata (e.g., transaction file system, versioning file system) makes logging no longer low-latency.

[1] Soft updates [15] and Patch model [16] study a consistency policy only for file systems whose users will face the persistence dangers that they may choose to accept (e.g., potential for loss of 30 seconds worth of information in most UNIX-derived systems), which is not the main focus of this project.

- Object Update does not break the logical locality of objects unlike Coupled Address Update.
- Coupled Address Update changes the mapping relationship between the object ID and the object address. In a system with complex data structures, it may cause a *recursive update* problem and compromise the system concurrency. Recursive updates are expensive address updates when modifying object in complex data structures (*e.g.*, index tree). This is because the value of object's address (*i.e.*, object pointer) and all relevant control structures (*e.g.*, metadata in file systems) need to be changed.

Furthermore, existing solutions employ a *1-1 mapping scheme* to implement the addressing function, which translates one logical address into a unique physical location to locate the requested object. As a result, an object updating system manages object and its copies in *dispersed storage spaces*, which imposes extremely high overhead about maintaining different copies. A coupled address updating system does not write object twice by migrating object to new locations.

Our solution to ensure storage consistency is based on a new observation of the addressing function, which can be leveraged to resolve the aforesaid limitations. We develop a new address update mechanism, called **decoupled address update** using *1-n mapping scheme*. The 1-n mapping scheme maps one logical address to two or more physical addresses, which renders two important features. First, we manage the object and its copies in a *unified storage space*. Hence, clients access the new object without physically overwriting the original object and avoid writing it twice. Second, we decouple the logical address update from the physical address update. Upon an object update, the logical address of the object is kept unchanged while all changes to physical locations are handled transparently to upper layers by 1-n mapping scheme. As a result, there will be no recursive updates. In summary, our new mechanism is able to ensure a consistent system state while maintaining high performance and without writing object twice and incurring recursive updates.

The main contributions of this work are:

- We analyze pros and cons of current solutions to ensure data consistency in file systems. We conclude that, existing one-to-one addressing function is one of the major components that give rise to the aforementioned limitations.
- We develop a new address update mechanism called as *Decoupled Address Updates (DAU)* that employs a one-to-two/many addressing function to realize benefits of both object update and coupled address update mechanisms and avoid most of their limitations.
- We implement a prototype system called *Kero* to conduct performance evaluation studies by comparing with several representative file systems including Ext2, Ext3, ReiserFS for both file system and database-like workloads. Comprehensive experimental results indicate that Kero significantly outperforms the baseline systems in most cases.

2 Related Work

Consistency in file systems: A myriad of logging algorithms have been developed to guarantee both data and metadata consistency in file systems [24][25][26]. The last decade has seen several recognized works in resolving the metadata update problem [15]. Most local file systems, such as Ext3 [20], JFS [17], XFS [18], etc. adopted the WAL algorithm, so-called journaling, to protect metadata consistency. Some local file systems also provide options to log both data and metadata for strong consistency requirement, such as the Ext3, Log-structured file system [27]. In log structured file system, a garbage collection procedure is expensive, it needs to recollect all the invalid records and assemble all the valid records in a file to a continuous zone. In Kero, garbage collection incurs less overhead because the log size is smaller, and it can access both the file system and log space. Table 1 summarizes the consistency schemes in file systems.

Similar to databases' shadowing approach, NetApp's WAFL [19] develops an online backup storage system that can be quickly accessed with a coupled address update mechanism. After that, most recent local file systems including Ext3cow [5] and ZFS [4] adopt COW to better support versioning and deliver high I/O performance. Ext3cow uses snapshot of the whole file system at various timestamps, hence all the updates between the crash point and latest timestamp are not reflected. In ZFS, updating any data block will cause a recursive procedure. If a client just updates one block in a large file which has multiple-layered indirect blocks, this mechanism would cost lots of extra write operations.

ReiserFS [22] considers recursive update problem in its implementation and tries to update data block just once. If the data in a node and its parent is changed in one transaction, reiserFS uses COW mechanism to update the node which saves one write operation to the disk. But if the parent node is not changed, reiserFS records the data node in a log to guarantee the atomicity and durability of the transaction. In this scenario COW mechanism makes the parent node dirty and causes the recursive update problem. Our decoupled address update mechanism avoids recursive updates by using one-to-many address mapping function.

Recent years have seen several novel methods to realize consistency in file systems, driven by a trend of increasing complexity of data storage applications and computer systems. Soft updates [15] is a technique different from the journaling by trading durability for better performance with consistency guarantee [28]. Inspired by soft updates, a patch work [16] develops a model to generalize all file system dependencies. It simplifies the implementation of consistency mechanisms for solving the metadata updates problem

Table 1. Consistency techniques in file systems

Techniques	Pros	Cons	Examples
Journaling/Logging i.e., object update [20, 21]	complete history, high concurrency, no atomic block write assumption	write object twice, require crash recovery	ext3, JFS, XFS, NTFS DBFS
Extended Journaling [22]	write object once if its parent node is also updated	logical log, similar to journaling	Reiser FS
COW i.e., coupled address update [19, 5, 4]	no redo/undo, no need for crash recovery	recursive updates, low concurrency, fragmentation	ZFS, WAFL, System R, ext3cow
Soft Updates [23]	reducing synchronous writes	lack of durability, non-negligible mem. req.	UNIX FFS
Patch [16]	general model, handle write-before relationship	limited performance	any FS
Kero (i.e., decoupled address update)	write object once, no crash recovery, high concurrency	some mapping overhead	any FS

in the file systems by separating the specification of write-before relationships from their enforcement. In short, journaling/logging has been recognized as a low latency mechanism to provide consistency, however the amount of data and metadata in the current systems demand to revisit the existing solutions.

Performance studies of Object Update and Coupled Address Update Mechanisms: Researchers have also investigated the I/O performance impact of both object updates and coupled address updates techniques. Rosenblum and Ousterhout developed a log-structured file system [27] that writes all modifications sequentially to disk in a log-style structure. Its large sequential write makes good use of nearly all of the disk bandwidth and thus obtains high write I/O performance. Because it is the first file system totally using coupled address update, many works further investigated its merits and shortcomings[29][30][31][32]. It is concluded that garbage collection is the big stumbling block to success.

Logical disk inherits the rationale of log-structured file systems by mapping file system to disks using a log-like structure [33][34]. It defines a new interface to disk storage that separates file management from disk block management. Similar to the logical disk, a more recent work named type-safe disk [35] realizes better security, integrity and semantics-aware performance optimization with awareness of the pointer relationships between disk blocks imposed by higher layers such as file systems. We can use type-safe disk to implement Kero efficiently. Another interesting work is hFS [36], a hybrid local file system that adopts a combination of two policies. hFS manages metadata and small files in a log partition while stores large files in a normal data partition. Their experimental results show reasonable performance gains. It shares an idea of the hybrid approach with one specific Kero implementation in local file systems.

Table 2. Notations

Notations	Meaning
S	Various spaces in a computer system. $S_{logical}$ denotes logical storage space; $S_{physical}$ denotes physical storage space; S_{shadow} denotes shadow space; S_{Kero} denotes the a part of logical space used by decoupled address update; and S_{backup} denotes backup space.
obj	An object in logical storage space.
$unit$	An atomic storage portion in physical storage space.
$meta$	Metadata object.
v	A data value. To obtain the value of an object, function $V(obj) = v$ is used.
$F(obj)$	An addressing function for mapping an obj to an $unit$.
$D(unit)$	The function retrieving an object from an $unit$.

3 Overview of Existing Update Mechanisms

File systems either use **Object Updates** or **Coupled Address Updates** to achieve consistency. Both solutions use a *1-1 mapping scheme* to implement the addressing function, which translates one logical address into a unique physical location to locate the requested object. We will briefly describe the addressing functions, their properties, and state-of-the-art update mechanisms with their limitations in this section.

Addressing Functions: An *addressing function* maps a logical object to a physical storage unit. For instance in Linux file systems files are interpreted as a set of logical block numbers (LBN), and each LBN is used to locate a file block.

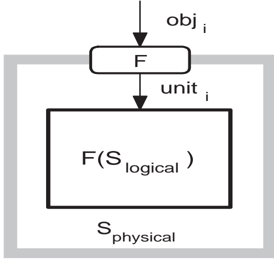

Figure 1. Address translation in a File System. $F(S_{logical}) \rightarrow S_{physical}$

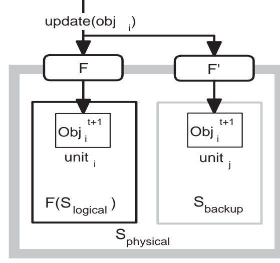

Figure 2. Address translation in an Object Update mechanism.

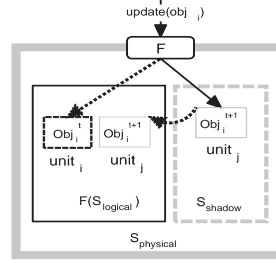

Figure 3. Address translation in a Coupled Address Update mechanism.

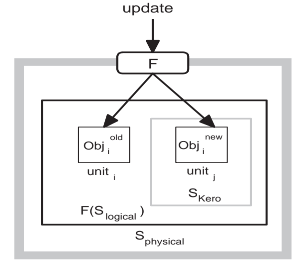

Figure 4. Address translation in the Decoupled Address Update mechanism.

Figure 1 illustrates the relationship between $S_{logical}$ and the addressing function in a file system. It is noted that all address translations in current file systems are denoted by one-to-one mapping, which is called **unique address property**, i.e. there exists one and only one $F(obj_i)$ for any object obj_i. Note: All the symbols are explained in the Table 2.

Furthermore, addressing functions are classified into two categories: *Direct Addressing Functions* (DAF) and *Indirect Addressing Functions* (IDAF). DAF can be a linear or hash function. On the contrary IDAF can not use an object to compute the address directly. It needs to get the value, *e.g.* a pointer, of another object beforehand. In IDAF, modifying an object address results in updates on other objects.

For example in a B^+ tree based addressing function, to access object obj_i in a block, the system must access all the indirect nodes on the path from the *inode* to the obj_i. Hence, if the file system organizes the metas in a tree or a list structure, the update operation in IDAF will be recursive, and this important property is called **chain updates property**.

In our new decoupled address update mechanism, we aim to exploit one-to-one mapping to create one-to-many mappings. It is anticipated that this will allow us to change the unique address property, and as a result the recursive updates can be avoided.

Object Update Mechanism: In an Object Update mechanism, the address of an object remains unchanged, and the new value of obj_i replaces the old one in $unit_i$. In a file system, an update request from an upper layer client consists of two steps.

(1) Write the new object $obj_i.v$ at a different address $unit_j$ as a backup, i.e. $write(unit_j, obj_i.v)$.

(2) Write the new object $obj_i.v$ to the same address, i.e. $write(F(obj_i), obj_i.v)$.

It is noticed that step(1) does not trigger the system state transition as it is performed to backup the consistent state of the system. The system transitions to a new state only after the object is updated as shown in step(2). To keep the system consistent, step(1) should always precede the step(2). If there are multiple inconsistent objects, we should ensure step(1) of all inconsistent objects finish execution before starting executing step(2) of any inconsistent object. Otherwise the system will transition to inconsistent states. Journaling is a typical object update scheme. Figure 2 illustrates this scheme. S_{backup} contains objects as backup only for the purpose of crash recovery. It does not belong to $S_{logical}$, and therefore can not be used by clients. In this scheme $unit_j \in S_{backup}$ and $unit_j$ is managed by another addressing function F'.

Coupled Address Update Mechanism: In the Coupled Address Updates, the original mapping unit $unit_i$ of the object obj_i is replaced by $unit_j$. It is done by changing the pointer in $meta_i$. Hence, the steps are:

(1) Write the new object $obj_i.v$ at a different address $unit_j$, i.e. $write(unit_j, obj_i.v)$. Note that this step is same as for an object update

(2) Update $meta_i$ as a result of this address update, i.e. $write(F(meta_i), unit_j)$.

We can see that updates in the coupled address update mechanism results in recursive procedures. Generally, it would end by writing a master meta (*e.g.* inodes in file systems) to the storage. Assuming that write is an atomic operation, coupled address update can achieve consistency because step(1) commits all the new values, but the system state does not change until the master meta gets written.

COW is an example of this mechanism. Figure 3 demonstrates the conceptual diagram of decoupled address update. S_{shadow} contains shadow version of objects in $S_{logical}$. After modifying metas, S_{shadow} becomes part of $S_{logical}$ and thus can be accessed by clients directly. This procedure is called *space switch*.

Limitations: Both mechanisms have shortcomings. For the object update mechanism the data in S_{backup} can not be used directly. Although the valid data is already in S_{backup}, the storage system still needs to update it at $unit_i$ again to make it available to the clients. On the other hand, if there is a crash during step(2) in object update, $S_{logical}$ transitions to an inconsistent state. A recovery routine is needed to retrieve a consistent state from S_{backup}. The procedure is:

$write(unit_i, read(unit_j))$, writing objects twice and performing crash recovery would degrade system I/O performance to some extent.

Coupled address update avoids the shortcomings of the object update mechanism. By storing a copy of an object at a different address in logical storage space, it avoids redundant writing. Furthermore, although step(2) in coupled address update usually needs to perform several I/O operations, the state transition procedure usually requires modifying one meta. Given the assumption that this update is atomic, coupled address update can guarantee that $S_{logical}$ is always consistent. As a result, no crash recovery procedure is needed. However, all above-mentioned advantages derive from IDAF. It is known that the coupled address update mechanism causes chain updates in the storage system. An address update of the object *i.e.* meta update would lead to an update in the address function $F(meta_i)$ itself. These associative updates of constitute a recursive behavior as follows:

$while(i! = root)$

$write(F(meta_i), unit_j)$ where j corresponds to the parent node of i, and $unit_j$ stores the value of $meta_i$.

In addition to the recursive updates, the system concurrency is affected because lots of related objects are locked even if only one object is to be modified. Lastly, a logical object is divided into several non-contiguous pieces in $S_{physical}$. This would break logical locality and incur a fragmentation problem in the system.

4 Kero: A Consistency Scheme using DAU

We develop Kero, as a high performance consistency scheme using a new Decoupled Address Update mechanism (DAU). Decoupled address update mechanism is developed using a *1-n mapping scheme*, which maps one logical address to two or more physical addresses. Its main purpose is to avoid the limitations of existing update mechanisms yet providing the same consistency semantics, i.e. object update and coupled address update. These limitations include writing the object twice in object updates mechanism and recursive updates in coupled address updates. We manage the objects and its copies in a *unified space*.

In this paper we propose a *novel addressing translation mechanism* that solves the above mentioned problems. Unlike, a coupled address update where an update results in modifying both logical and physical address, it decouples the logical address update from the physical address update. Upon an object update, the logical address of object is kept unchanged while all changes to the physical locations are handled transparently to upper layers by 1-n mapping scheme. Kero space S_{Kero} is a part of $S_{logical}$, and is used in our new mechanism as shown in the Figure 4. We call our new update mechanism as **Decoupled Address Updates**, which implements a new addressing function by mapping one obj_i to either $unit_{i0}$ in the $S_{logical}$ or $unit_{ix}$ in the S_{Kero} as shown in eq. 1.

$$F(obj_i) = \begin{cases} unit_{i0}, & S_{logical} \\ unit_{ix}, & S_{Kero} \end{cases} \tag{1}$$

It should be noted that in the $S_{logical}$, we perform an object overwrite; in S_{Kero}, we write to a different unit $unit_{ix}$, which could accommodate the object value at any time point. If there is an update request, then based on a decision making rule in specific implementation, we either choose to update the unit in the $S_{logical}$ or we create a new unit in the S_{Kero}. We can see that the object is written once to the one of the mapped units, hence unlike object updates it does not have the "multiple writes" problem, and also avoids the meta updates in coupled address update.

5 Design Considerations

In this section, we discuss different design issues we considered in order to implement decoupled address update mechanism. We also include various design options and our preferences, especially for the one-to-two mapping function for decoupled address update, and minimizing the size of Kero space.

5.1 Decoupled Addressing Function

The fundamental issue in our design is how to implement a decoupled addressing function. This choice would influence all the other design options. The conventional computer systems do not support a decoupled addressing function, so we need find a new method to construct it. One way is to define a *1-1 mapping* that takes a **mark** as a new parameter: $F(obj, mark) = address$, to emulate the behaviors of *1-2 mapping* function. **mark** is introduced in the Section 4 Depending on the value of mark, either 1 or 0, this function can compute two different physical addresses for one logical address. For example, in the file systems with the *inode* structure, we can insert the mark parameter into inode. This approach needs to modify the original addressing function directly, which results in modest modification of the storage system.

Another approach uses two functions to provide 1-2 mapping. The first function is the original addressing function, i.e. $F(obj) = address$, and the address is taken as an obj_i in the second function as: $F(obj_i, mark) = address$. A practicable implementation is to add a new mapping layer between the upper layer file systems and disk drivers. This approach obviates the need of changing the addressing functions in the upper layers, and the second function can easily be implemented in the mapping layer. In addition, this approach is independent of any specific file system, so it can be used to support multiple file systems. However it would introduce an extra overhead compared with the first approach. In our prototype, we use the second approach. It simplifies the programing and can be used to evaluate several different file systems as compared to modifying the addressing functions for each individual file system.

5.2 Minimizing the size of Kero space

Ideally the size of Kero space should be equal to that of logical space in our model. However, this might not be acceptable in most of the cases, even Kero provides impressive data consistency service and performance. In real applications, it is unnecessary to use half of the disk space for Kero (It maps 1 logical to 2 physical addresses). Firstly, it is nearly impossible that all the files are updated at the same time. In this scenario, even COW mechanism should keep half of the disk space for storing the shadow copies. Secondly, *Kero space can be reused* by implementing a low overhead garbage collection, when data is updated back into the logical space. Therefore, a relative small storage space is enough for Kero model. How to manage a small Kero space without undermining the Kero mechanism is a challenging task. In general, people attempt to map a large space to a small one by using hash functions. In case of conflicts, all objects with the same hash value are organized in a linked list. However, this leads to random access because the contiguous blocks in logical space may spread out in the storage media, especially on disk, which degrades system performance greatly. In our implementation, we choose to use a 2D-array as a mapping table because we want to write contiguous blocks on the disk.

5.3 Enforcing the atomicity of updating a group of objects

In our design, Kero uses DAU mechanism at block layer with a mapping table. Then the consistency between mapping table and two spaces is a basic condition that we should guarantee. There are two approaches to keep the updates of objects atomic. A simple way is to allocate two separate spaces in the storage to save them. Each space is assigned a logical timestamp. After all the updated objects are synchronized to the disks, a new timestamp is recorded by incrementing the current timestamp by one. The space with a more recent timestamp stores valid object. The other approach is based on a log-structured Kero space. Given a log, we are able to write the objects to the log space directly and utilize the properties of log to enforce the atomicity of objects updates.

5.4 Crash Recovery

In the ideal scheme, the file system state transition is done by writing a file mark. Then assuming that this single bit writing is an atomic operation, crash recovery is not needed. However, because we try to save storage space and improve I/O performance, a mapping layer and a mapping table are used. The information in the table is crucial for the file system consistency and it should be kept persistent. On the other hand, we implement Kero at block layer which means several marks should be updated in one file update operation. Therefore, a crash recovery is still needed. But compared with crash recovery in journaling file system, the procedure is very simple because Kero just recovers a table which contains much less information than journaling. What's more, in most cases, only one table entry is revised when a file is updated. The atomic block update assumption is common in file systems, and therefore crash recovery is unnecessary.

6 Implementation

In this paper we focus on the implementation of decoupled address update in a unified space as a module between the Linux file systems and generic block layer. In this section we associate our Kero model with file systems. With Kero the storage system can separate the complicated addressing structure from the exact physical location of the data. It can provide persistent data consistency with high I/O performance because it saves one object write for large disk updates; and inherits low-latency logging benefit for small updates. Figure 5 illustrates a conceptual architecture of our design.

6.1 Implementing the Kero Space-S_{Kero}

Normally the logical spaces of file systems are managed with tree structures(Linux file systems) or linked list structures(FAT file systems). The file data uses object update in these systems. Those structures work fine in Kero space. However, Kero manages its storage space in a log-structure fashion, henceforth, named semantic log. All the data being written to the Kero space would be written to disk contiguously, just as a regular log service does. The difference between semantic log and regular log is that the data on the semantic log can be accessed by the clients with the supports of 1-2 decoupled function. There are several merits to build Kero space as a log. Firstly, without changing the original data objects, writing the updated objects to storage with a log manner can guarantee the atomicity and the durability of the operation. With the supports of log, we can simplify the implementation greatly. Secondly writing data to disk contiguously can improve the system I/O performance dramatically. Thirdly, the log structured space can be reused easily.

When an upper layer file system updates data, and there is no related information in the mapping table, the mapping layer redirects the request to the log and then records the LSN of this record. Hence from the upper layer file system's perspective, it is an object update policy because the data addresses are unchanged. A direct benefit is, even for a system that employs complicated data structures to support fast indexing, both recursive update and low concurrency problems are resolved. From the underneath layer's perspective, the copy of new data is written at a shadow address, which means a coupled address update. By recording the data address in the Kero space, the system does not have to update the object again.

In the real implementation, Kero utilizes a table to map physical address in file system space. Each in-flight blocks which is mapped into log record has one entry: (address, LSN)(LSN is log sequence number). When a block is updated back into file system space, the related entry is re-

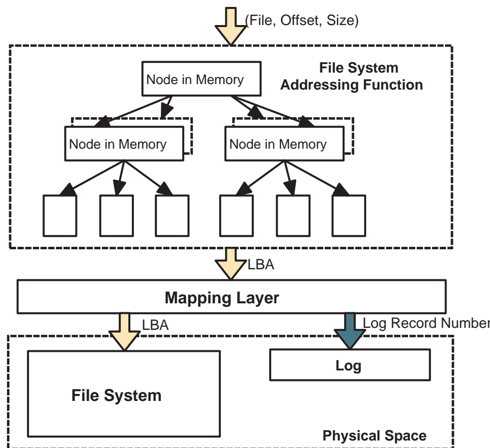

Figure 5. This figure demonstrates the architecture of our design. The light arrow represents a flow controlled by file systems. They can only access part of the storage space. The mapping layer implements a table that records the original data address in the file system and a shadow address of the data in log.

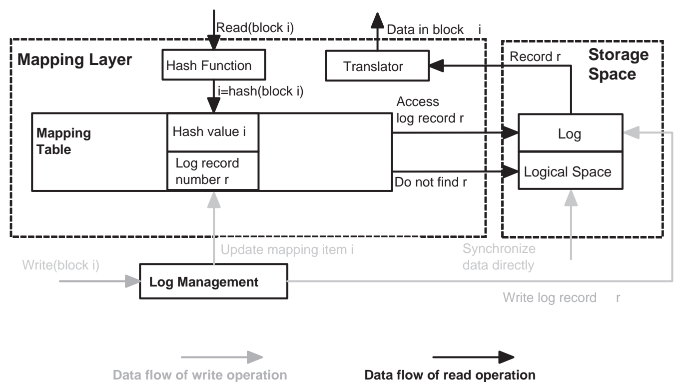

Figure 6. This figure demonstrates the components of the mapping layer and how it works.

moved from the table. By using this table, the mapping layer can keep track of the mapping address of an block in Kero space.

6.2 Implementing the Addressing Function

As discussed in the Section 5, we implement the addressing function by using a mapping layer. Some system resources such as internal memory space and CPU time are consumed in its implementation. It would potentially degrade the overall performance if this layer incurs much overhead. An important question to answer is how much mapping information we should maintain in this abstract layer. First, if a piece of data does not have a shadow version, Kero does not need to create an entry for it. We only need to keep the mapping information for those blocks which data are not synchronized to the file system yet. Just as aforementioned, our system sometimes chooses to update data in its original block address to promote logical locality. It is unnecessary to store the mapping information for such data. Additionally, because all the victim pages are flushed into the file system, the pages not present in the page cache do not need to be recorded. In our prototype system, we implement two fields — LBA (Logical Block Address) in file systems and the address of log record per mapping table entry, each of which is 4 Bytes. In the worst case, Kero needs $8/4K = 0.2\%$ of the total page cache size to store all the mapping information.

Based on the above analysis, we simply choose hash functions to do the mapping job. Since hash is a DAF, this would avoid recursive updates problem. Given a common LBA in the file system, the hash function calculates a hash value, which becomes an index for the mapping table. The other field in the table is the address of the corresponding log record. After computing the index, Kero looks up the table. Upon a hit, the data in the log is accessed. Otherwise, Kero accesses the data from the file system.

There are two kinds of recovery strategies, depending on whether we make the mapping table durable or not. First, if the mapping table is not stored on disk, crash recovery is needed when the system reboots from crashes. However, during the normal process, redo and undo functions are not needed. Second, if we store the mapping table on external memory, the file system does not require a crash recovery process. In our experiments, we implement the second approach. There are multiple dedicated segments to store the mapping table in the log. When Kero performs some writes on log, the nearest segment is chosen to store the mapping information. To identify which segment holds the most recent data, a timestamp is used. We make the table durable by two steps. The first step is to write the table content to the log space. This is done by packing the table as a log record. Furthermore, the log record is assigned a logical timestamp t at the time point. After the table is written to the log, a synchronous write request gets executed by Kero. This operation writes t into a specific address in the segment, which is similar to writing a commit record into log in the WAL algorithm. If two time stamps match, the segment contains valid data; vice versa. When the system reboots from crashes, the valid segment with the largest time stamp holds the most recent valid table. With this table and data in both file system and log, Kero can rebuild a consistent system state before crash. The diagram of the mapping layer is illustrated in Figure 6.

Decision Making Rule in Kero: A decision making rule in our implementation specifies whether to update the unit in the file system or create a new unit in the log as shown in the Figure 5. Most of the local file systems often adopt object update to maintain logical locality. They save the related data in a continuous space. An exception is log-structured

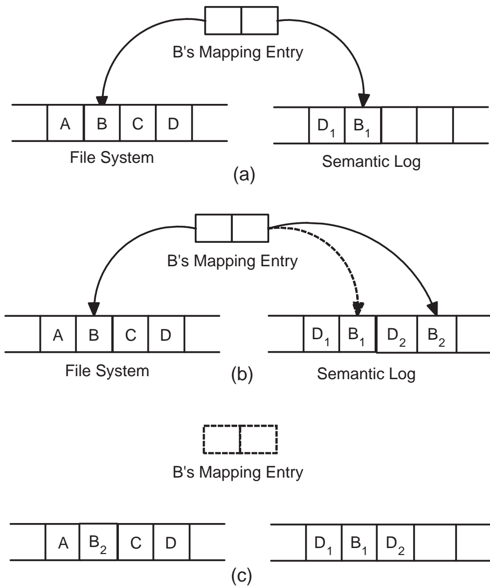

Figure 7. This figure demonstrates the way that Kero writes new data to a block. Graph (a) shows a initial state of system. Graph (b) shows if the block B is a hot block, which means the time internal between two block operations is less than the pre-defined time-out variable, when a write operation comes, Kero would write the new data into log space to keep temporal locality. Graph (c) shows that if the block is not accessed for a long time, exceeding the pre-defined time-out variable, Kero redirects the write operation back into the file system, and the corresponding mapping entry is abandoned. If the mapping entry for a block is not found, that means the latest copy is in the file system.

file system [27] that employs coupled address update to obtain temporal locality. But neither of them realizes both advantages. Kero can achieve both localities to some extent because it can access both file system and semantic log. To implement the semantic log, we define a timeout variable, which captures the degree of temporal locality.

More specifically, Kero writes the data into a log-style space. After the log records are written to the disks, the immediately following reads are directed to the log space. If the upper layer systems do not access a file for a long time, it implies that system would not access these data any more in the near future. At this point, we can update these data in the file system. This approach promotes logical locality. To implement this scheme, we do not switch two spaces right after a file is updated. Kero writes data back into file system space only when it is not accessed for a long time. When the storage system flushes dirty data out of the page cache, all the last-access-time fields in the mapping table would be checked. If the result of subtracting the flush time to the last access time of corresponding files is larger than the timeout, the mapping layer would dispatch the request to the file system. By adopting a new hybrid mechanism to achieve both kinds of localities, Kero can improve system read performance to some extent. Figure 7 demonstrates two situations of update operations in Kero.

On the other hand, the page cache would accommodate many small in-flight writes and then consolidate into a large sequential one. Hence, when Kero updates the file system, we defer the requests for a while to merge several small I/O operations into a single large one. As discussed previously, a log is used to keep as much updates as possible in Kero. Then a large random write workload can be converted to a sequential one. As a result seek and rotational latency are reduced greatly. Because Kero can guarantee file system consistency with one write operation instead of two, system write performance can also be promoted for small random writes. Therefore, our implementation not only keeps the storage system consistent, but also potentially promotes the overall I/O performance in disk-based storage systems.

During system running, some log records are invalidated due to repeated block updates. As a result, a *garbage collection* process is needed. This process would migrate all the valid data back into file system and then release the whole log space for reuse. In our implementation, the overhead of this process is very small. Firstly, the size of log is very small. To guarantee file system consistency, Kero just needs several Mega bytes to save the records, similar to ext3 file system. Secondly, Kero manages file system space and log space together. As discussed before, when the system flushes the dirty blocks out of the buffer cache and a block is not accessed for a long time, it would be written to the file system rather than the semantic log. There are other situations that data are written back. For example, if a page is swapped out of the main memory, its content would be updated in file system because the block is not hot. Based on the above-mentioned facts, when the utilization space of the log is approaching to a threshold (e.g., 80%), all the updates to the blocks which have mapping entries are directed to the file system to save log space. By the time when Kero performs garbage collection, lots of information is written back to the file system already, and therefore the number of valid records is not large. In summary, the garbage collection procedure in Kero is efficient.

6.3 Implementing the Prototype System

The Kero prototype system runs as a Linux 2.6 kernel module. It interacts with the Linux generic block device layer. In general, each I/O operation executed in traditional file systems involves a group of blocks. In this layer, the requests dispatched from the upper layers are reorganized in the unit of disk sector. Our Kero module replaces the generic block layer with the new one-to-two address mapping function. By introducing a mapping table, one block can be addressed with two different set of sectors.

Kero consists of a set of dynamic Linux modules. It provides several functions for the Linux kernel. At first we revised the Linux kernel to provide a set of new interfaces for Kero. This is implemented by declaring many new

structures of function pointers in the Linux kernel source code. During the initialization procedure of modules, Kero sets several signals and assigns valid values for all function pointers. When the kernel functions snoop the relevant signals set, they switch to the corresponding functions that Kero pre-defines. After that, Kero functions take over the program control.

Kero modules interact with Linux generic block device layer mainly via the submit_bh function. In main memory, the block is the basic unit of data transfer for the VFS, while the sector is the basic unit of data transfer for the hard drive device. This function is little else than a glue function that creates a bio structure from the content of the buffer head and then sends read or write requests to a Linux disk scheduler, which may eventually release them to the device. By revising the submit_bh function call, the content of bio is decided by Kero. Each time when Linux generates a read bio structure, Kero checks if there is a mapping item in the table for the corresponding buffer head. If the answer is yes, the bi_bdev field is assigned with the block device description of our log device in the implementation. The bi_sector field is also set to the mapping location. After the data is read from disks, a record interception module is called. If it is a write operation and no item in the table is found for the block, the data is assembled into a log record and flushed to the log partition. The call back function notifies Kero to record the mapping information in the table. If a mapping item exists, the due time of the block gets checked. Data that are not accessed for a certain amount of time is object update otherwise coupled address update. An exception is when a page is removed from the page cache in the Linux page cache implementation. We revised the remove_from_page_cache function to remove the mapping items of the page in the table.

In our prototype, we implement decoupled address update at generic block layer in Linux with a log. This implementation not only enjoys all the merits of DAU, but also provides the same consistency semantics as ext3 (including cross-object consistency such as updates among directory block and file inode block). By recording different objects in the log, the Kero system can provide the same function as write-back journaling and full journaling. With the support of ext3 file system, a consistent model as ordered journaling also can be implemented.

Our modules implement all necessary functions as required in our original design. The new code is with nearly 2000 lines. We did not apply some popular optimization techniques, such as extent based addressing translation, group commit and so on. It is anticipated that Kero can achieve better performance by incorporating more popular performance optimization schemes.

7 Evaluation

In this section we evaluate the effectiveness of Kero, the performance of the Kero prototype system relative to different baseline file systems including Ext3 and reiserFS using both file systems and database-style benchmarks.

7.1 Experimental Setup

All experiments were performed on a commodity PC system equipped with a 3.6 GHz Intel E5320 dual core processor, 1 GB of main memory, and one 7200 RPM Seagate SCSI disk of 250 GB capacity. In the tests we used a 50 GB file system and a 20 GB log partition with Linux 2.6.11 kernel with the Fedora v4 distribution. Log partition was left empty. And only 4MB capacity is used for each Kero prototype. To reduce the influence of other non-related applications in the experiments, we disabled as many system services as possible.

To conduct fair comparisons with the state-of-the-art, we implemented two new "journaling" file systems on the upper layer of Kero. We tested the performance of both Kero prototype systems with three other baselines. All the file systems we tested are listed in Table 3. In ext3 with an *ordered model*, only updates made to file systems metadata are logged into the journal. The ext3 file system groups metadata and relative data blocks so that data blocks are written to disk before the metadata. On the contrary, in a *full journal model*, both file system data and metadata updates are logged into the journal. This model minimizes the possibility of losing the updates made to each file. When we incorporate a file system with Kero, asynchronous model of the file system is always chosen. Because the Kero prototype system monitors all the data updates at block level, enforcing ordered updates between metadata and data is not necessary any more.

Table 3. File system notations used in experiments

Notations	File Systems.
ext3(o)	Ext3 with an ordered journal model.
ext3(f)	Ext3 with a full journal model.
reiserFS	ReiserFS file system.
ext2-k	Ext2 with Kero support.
reiserFS-k	ReiserfsFS that employ Kero instead of its own journal service. This is done by using mount options.

7.2 Methodology

To comprehensively evaluate Kero, we chose three different kinds of benchmarks. The first postmark benchmark emulates a heavy small file workload seen on email servers, net news servers and web-based commerce. We used PostMark v1.5 [37] in experiments, and configured it to create 1,000 files ranging in sizes from 512 B to 1 MB. Other configurations include performing 1,000 transactions inc. both reads/writes and creates/deletes. The read or create bias parameter is set to 5.

Bonnie++ [38] is a second benchmark suite that aims at performing a number of simple tests of hard drive and

file system performance. The first part of tests came from the original Bonnie program. It initially performs a series of tests, such as sequential output, rewrite, sequential input and random seek, on a large file with 1 GBytes in our experiments. The performance of per-character sequential output and input is not tested in our experiment. The next six tests involve file create/stat/unlink to emulate some operations that are common bottlenecks on large Squid and INN servers. More specifically, there are different workload patterns: sequential create, sequential read, sequential delete, random create, random read and random delete. Lastly, we vary file sizes in the last six tests to evaluate the CPU utilization rate of file systems. In all the tests, for performance, the higher of the number, the better; for CPU utilization rate, the lower, the better.

The last benchmark is a modified TPC benchmark, which was used to evaluated the performance of log-structured file system [27]. We do not use the new TPC benchmark suite because most of them portray database applications under distributed environments. However our experiments focus on the performance of local file systems and databases. This benchmark emulates a check-cashing application. There are four files in our configuration: an account file contains 1,000,000 records; each record is a 1,000-byte structured data; a branch file has 10 records while a teller file has 100 records. We use this micro-benchmark to evaluate the I/O performance of local file systems when they execute lots of concurrent requests. File systems deployed in the Linux environment always lock a file when a process accesses it. To replay a reasonable emulation environment, we group eight records (8 K) into a file to provide a similar grain level with database systems.

7.3 Results

PostMark: Figure 8 shows the overall performance of five different file systems. This set of tests focus on evaluating the I/O performance of file systems under small random requests. Since Kero provides the consistency control mechanism for the upper layer file systems, we are able to buffer a large number of writes in memory such that file creates and deletes become asynchronous operations in Kero enabled file systems. As expected, the asynchronous pattern and the sequential writes of Kero yield superior performance to other file systems for PostMark. By enforcing a same level of consistency, Kero spent only 46% execution time of ext(f) to finish all the tests of postmark. Ext3(o) guarantees less consistency than ext3(f) and ext2-k, but ext2-k achieves a same performance as that of ext3(o) in both create and delete phases, and spends 30% less time on the transaction phase. ReiserFS performs well for small reads and writes. The current reiserFS version obtains comparable performance with our revised one with Kero component. However, in the create and delete phases, Kero wins by about 40% performance improvement.

The transaction phase of PostMark consists of lots of read and write operations. Figure 9 shows the read and write

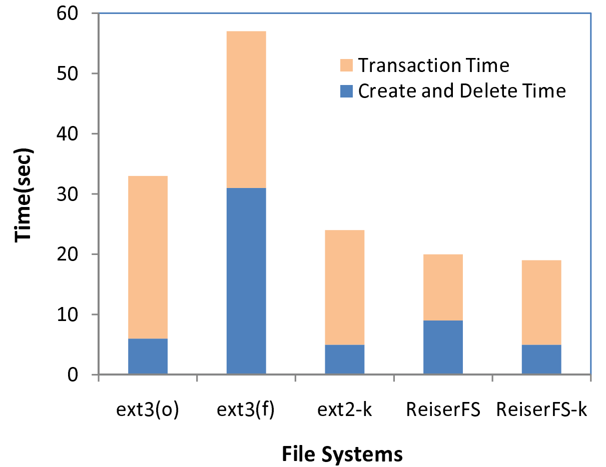

Figure 8. The graph displays the performance of the overall tests of PostMark. Kero performance dominates ext3 file systems series because the workload consists of operations on small files and Kero can make them sequential. ReiserFS, compared with others, provides good transaction performance because the high bandwidth of small read and write. However reiserFS-k outperforms it in the create and delete phases. The log-structured layout boosts I/O performance.

bandwidths of different file systems. We can observe that if we want to ensure consistency as that in a full journal model, ext2-k outperforms ext3(f) on writes by nearly 121%. On the other hand it obtains a comparable read performance to that of ext3(o). In the reiserFS series, the read performance of reiserFS-k is close with that of original reiserFS. Although reiserFS just provides logical logging service and in some specific scenarios it only updates data once, the write performance of reiserFS-k is still approximately 16% better than that of original reiserFS.

Bonnie++: Different from PostMark that focuses on testing small files operations, Bonnie++ benchmark tests small I/O operations on a large file. Figure 10 demonstrates three different workloads in this environment. The first one is sequential output, which means the file systems create files using write. This is a file space allocation intensive workload. In this case, all five file systems except ext3(f) obtain comparable performance. Ext2-k achieves 94% better performance than ext3(f) and nearly 10% higher throughput than reiserFS. In the rewrite test, three commands including read, rewrite and lseek would be executed. File systems with efficient addressing function and high read/write throughput often obtain good results in the test. The graph shows that reiserFS's B^+ tree structure significantly promotes the performance of a file system in this test. Given the same kind of addressing function, ext2-k outperforms ext3(f) by about

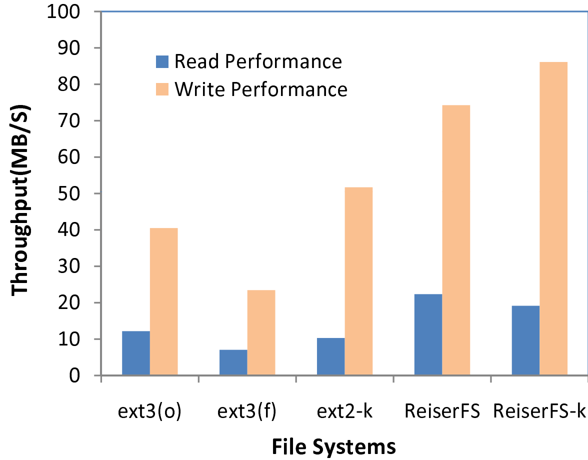

Figure 9. The graph displays the performance of read and write tests of PostMark. The ext2-k write performance is approximately 25% better than ext(o) and 121% better than ext(f) resulting from the sequential writes property. ReiserFS obtains good read and write performance for small files due to its fast addressing and efficient logical logging service. ReiserFS-k still outperforms it in write performance.

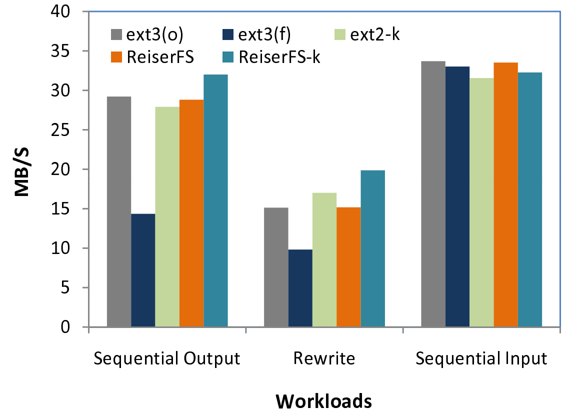

Figure 10. This graph shows the Block I/O Performance Results of Bonnie++. The main difference between sequential output and rewrite is that sequential output just creates a file using write while rewrite needs to perform read and lseek additionally. This experiment tests the effectiveness of file system cache and the speed of data transfer. The I/O performance of all file systems degrade dramatically when comparing the sequential output workload run with the rewrite workload run. Kero-k dominates in this test due to the combination of B^+ tree based addressing function of reiserFS and logging mechanism of Kero.

73%, and reiserFS-k outperforms reiserFS by about 31%. This is because Kero writes data only once in a sequential fashion. In the last case the read performance of Kero is a little bit lower than other systems. The reason lies in that, although some read operations are absorbed by in-memory page cache, Kero still has to execute lots of read requests. Its mapping layer introduces some overhead to the file systems and breaks logical locality with some data. However, all other baseline systems execute the read requests in a sequential fashion.

CPU Utilization: Table 4 and Table 5 demonstrate the results of CPU utilization rate for the last six tests of bonnie++ benchmark. Those workloads are CPU intensive due to lots of metadata operations being executed. We want to evaluate the overhead the new block layer addressing function would introduce, e.g., consuming CPU cycles. The tables imply that, first, Kero does not spend much CPU time for all the workloads except read; second, the CPU utilization percentage of Kero drops fast with an increasing file size. For metadata intensive operations, such as create and delete, Kero obtains better performance than ext3 with an ordered journal model, since the ordered journal needs to keep the write-before relationship between every pair of data and metadata. This leads to lots of synchronous writes that waste CPU cycles. For sequential read, Kero obtains comparable performance with ext3(o). Under a random read workload, Kero makes some negative impacts on system I/O due to the loss of some logical locality. However the tables show that when the file size is larger than 128 KB, the CPU utilization rate of Kero is still very small. ReiserFS realizes The best read performance, although it does not work very well under create and delete workloads. This results from its complicated logical structure of reiserFS that render more CPU cycles.

Highly Concurrent Workloads: The last experiment aims to evaluate the performance of file systems under highly concurrent workload, which is a common case in database applications. Four sets of results are shown in Figure 11. The performance of ext3 with a full journal model is the worst because ext3 file system has to record lots of random operations into the log partition. Hence I/O becomes the bottleneck and degrades the system performance greatly. In reiserFS, compared with Kero and ext3 with an ordered journal model, I/O is still a bottleneck. It implements a higher level granularity concurrency than other file systems in the experiments. Therefore reiserFS underperforms Kero and ext3(o).

Kero beats ext3(o) if the number of concurrent threads is lower than 300. This is because that given the light workload, space switching rarely occurs and Kero consolidates many small I/Os into a large contiguous one. Given a heavy workload, space switching occurs more often. In addition, because the log partition is nearly full, synchronizing logical space needs to get executed. Both of above-mentioned actions would degrade system performance to some extent.

Table 4. CPU usage for sequential workloads

	Sequential Create				Sequential Read				Sequential Delete			
File Size	ext3(o)	ext3(f)	reiserfs	Kero	ext3(o)	ext3(f)	reiserfs	Kero	ext3(o)	ext3(f)	reiserfs	Kero
[2, 4]	63	22	26	25	100	99	1	100	46	97	4	29
[4, 8]	29	15	44	17	99	98	10	99	52	99	39	19
[8, 16]	27	11	58	11	95	98	5	99	96	98	33	9
[16, 32]	18	9	29	11	94	51	5	54	96	98	26	7
[32, 64]	11	6	48	10	2	2	3	2	2	2	15	2
[64, 128]	8	5	21	6	1	1	1	1	1	1	11	2
[128,]	9	5	13	4	1	1	1	1	1	1	7	1

Table 5. CPU usage for random workloads

	Random Create				Random Read				Random Delete			
File Size	ext3(o)	ext3(f)	reiserfs	Kero	ext3(o)	ext3(f)	reiserfs	Kero	ext3(o)	ext3(f)	reiserfs	Kero
[2, 4]	59	22	26	27	99	99	1	100	40	100	3	31
[4, 8]	53	15	43	20	98	98	11	99	94	94	19	21
[8, 16]	27	11	53	12	100	100	6	100	79	100	19	18
[16, 32]	18	9	31	8	98	99	5	99	100	99	12	11
[32, 64]	11	6	50	4	5	99	3	9	6	85	10	6
[64, 128]	8	5	20	4	1	2	1	2	1	1	7	1
[128,]	8	5	14	2	1	2	1	2	1	1	5	1

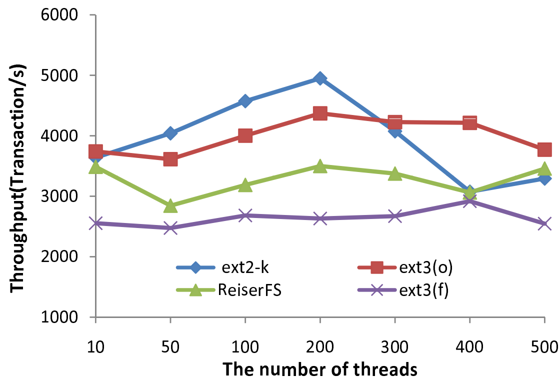

Figure 11. This graph demonstrates the transaction throughput of different file systems in the highly concurrent environment. ReiserFS obtains low concurrency because the B^+ tree structure limits the lock granularity. Kero achieves higher concurrency compared with ext3(o) and ext3(f). However when the number of concurrent processes becomes too large, the overhead of space switching and logical space synchronization would diminish the return of I/O performance gain and thus drag down the Kero system performance to a comparable level as other file systems.

Hence the I/O performance of Kero drops to a comparable level as other file systems when there are more than 300 threads.

8 Conclusion

In this paper, we develop a new data consistency scheme Kero that takes advantage of two existing update mechanisms, i.e. object update and coupled address update in file systems. We identify certain limitations in current solutions and discover one-to-one addressing function becomes one of the major obstacles to performance improvement. Kero attempts to manage data and its copies in a unified address space by employing a one-to-two/many addressing function. This renders efficient and effective data consistency maintenance and data management. By experimenting with several representative file system benchmarks, we conclude that Kero significantly improves I/O performance by upto 121% in comparison to several baseline systems—ext3 and reiserFS with different journal models.

References

[1] J. Yang, C. Sar, and D. Engler, “Explode: a lightweight, general system for finding serious storage system errors,” in *OSDI ’06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2006.

[2] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit, “Eio: error handling is occasionally correct,” in *FAST’08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), pp. 1–16, USENIX Association, 2008.

[3] B. Schroeder and G. A. Gibson, "Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you?," in *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, (Berkeley, CA, USA), p. 1, USENIX Association, 2007.

[4] "ZFS file system." http://opensolaris.org/os/community/zfs.

[5] Z. Peterson and R. Burns, "Ext3cow: a time-shifting file system for regulatory compliance," *Trans. Storage*, vol. 1, no. 2, pp. 190–212, 2005.

[6] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata efficiency in versioning file systems," in *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), pp. 43–58, USENIX Association, 2003.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, (Berkeley, CA, USA), pp. 15–15, USENIX Association, 2006.

[8] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, "Cache-oblivious streaming b-trees," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, (New York, NY, USA), pp. 81–92, ACM, 2007.

[9] Microsoft, "NTFS - New Technology File System designed for Windows Vista, XP, 2003, 2000.." http://www.ntfs.com, 2008.

[10] "WinFS." http://en.wikipedia.org/wiki/WinFS.

[11] B. Berliner and J. Polk, "Concurrent versions system (CVS)." www.cvshome.org, 2001.

[12] B. Liskov and R. Rodrigues, "Transactional file systems can be fast," in *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, (New York, NY, USA), p. 5, ACM, 2004.

[13] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok, "Extending ACID semantics to the file system," *Trans. Storage*, vol. 3, no. 2, p. 4, 2007.

[14] "Multidimensional database." http://en.wikipedia.org/wiki/Multidimensional_database, 2008.

[15] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft updates: a solution to the metadata update problem in file systems," *ACM Trans. Comput. Syst.*, vol. 18, no. 2, pp. 127–153, 2000.

[16] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang, "Generalized file system dependencies," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, (New York, NY, USA), pp. 307–320, ACM, 2007.

[17] "JFS overview: How the journaled file system cuts system restart times to the quick." http://www.ibm.com/developerworks/linux/library/l-jfs.html.

[18] "XFS: A high-performance journaling filesystem." http://oss.sgi.com/projects/xfs/.

[19] D. Hitz, J. Lau, and M. Malcolm, "File system design for an nfs file server appliance," in *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, (Berkeley, CA, USA), pp. 19–19, USENIX Association, 1994.

[20] "The ext2/ext3 file system." http://e2fsprogs.sourceforge.net.

[21] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system r database manager," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 223–242, 1981.

[22] "Reiser file system." http://www.namesys.com/v4/v4.html.

[23] G. R. Ganger and Y. N. Patt, "Metadata update performance in file systems," in *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, (Berkeley, CA, USA), p. 5, USENIX Association, 1994.

[24] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, 1992.

[25] R. A. Crus, "Data recovery in ibm database 2," pp. 322–332, 1986.

[26] A. Spector, R. Pausch, and G. Bruell, "Camelot: a flexible, distributed transaction processing system," *Compcon Spring '88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, pp. 432–437, 29 Feb-3 Mar 1988.

[27] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5, pp. 1–15, 1991.

[28] M. I. Seltzer, G. R. Ganger, M. K. Mckusick, K. A. Smith, C. A. N. Soules, and C. A. Stein, "Journaling versus soft updates: Asynchronous meta-data protection in file systems," in *In USENIX Annual Technical Conference*, pp. 71–84, 2000.

[29] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin, "An implementation of a log-structured file system for unix," in *USENIX'93: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 1993.

[30] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, "File system logging versus clustering: a performance comparison," in *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 1995.

[31] "A critique of seltzer's lfs measurements." http://www.sunlabs.com/people/john.outerhout/seltzer93.html.

[32] "A response to seltzer's response." http://www.sunlabs.com/people/john.outerhout/seltzer2.html.

[33] W. de Jonge, F. Kaashoek, and W. C. Hsieh, "Logical disk: A simple new approach to improving file system performance," tech. rep., Cambridge, MA, USA, 1993.

[34] "Atomic recovery units: failure atomicity for logical disks," in *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, (Washington, DC, USA), p. 26, IEEE Computer Society, 1996.

[35] G. Sivathanu, S. Sundararaman, and E. Zadok, "Type-safe disks," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, (Berkeley, CA, USA), pp. 15–28, USENIX Association, 2006.

[36] Z. Zhang and K. Ghose, "hfs: a hybrid file system prototype for improving small file and metadata performance," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 175–187, 2007.

[37] J. Ketcher, "Postmark: A new file system benchmark," *Network Appliance Inc.*, 2004.

[38] "bonnie++." http://www.coker.com.au/bonnie++/.