HIFS: History Independence for File Systems

Sumeet Bajaj
Stony Brook University
New York, USA
sbajaj@cs.stonybrook.edu

Radu Sion
Stony Brook University
New York, USA
sion@cs.stonybrook.edu

ABSTRACT

Ensuring complete irrecoverability of deleted data is difficult to achieve in modern systems. Simply overwriting data or deploying encryption with ephemeral keys is not sufficient. The mere (previous) existence of deleted records impacts the current system state implicitly at all layers. This can be used as an oracle to derive information about the past existence of deleted records.

Yet there is hope. If all system layers would exhibit history independence, such implicit history-related oracles would disappear. However, achieving history independence efficiently is hard due to the fact that current systems are designed to heavily benefit from (data and time) locality at all layers through heavy caching, and existing history independent data structures completely destroy locality.

In this work we devise a way to achieve history independence while preserving locality (and thus be practical). We then design, implement and experimentally evaluate the first history independent file system (HIFS). HIFS guarantees secure deletion by providing full history independence across both file system and disk layers of the storage stack. It preserves data locality, and provides tunable efficiency knobs to suit different application history-sensitive scenarios.

Categories and Subject Descriptors

H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*; D.4.3 [**Operating Systems**]: File Systems Management—*File organization*

Keywords

History Independence; File System; Secure Deletion

1. INTRODUCTION

Numerous regulatory frameworks in finance, government and health-care, require secure deletion assurances typically by overwriting records on deletion. However, simply overwriting records does not guarantee deletion since the write history itself implicitly leaves artifacts in the layout of the resulting storage medium at all layers. The artifacts can be used as an oracle to answer questions about the past existence of deleted records. For example, the current layout of data blocks on disk is a function of the sequence and timing of previous writes to file system or database search indexes. Questions such as "was John's record ever in the HIV patients' dataset" can then be answered much more accurately than guessing by simply looking at the storage layout of the search index on disk (which will look different depending on whether John has previously been in the data set or not). Yet, these are the very questions that secure deletion promises to prevent anyone (including insiders) from answering once John's record has been deleted.

We posit that if all relevant system layers exhibit *history independence*, the implicit history-related oracles would disappear making the recovery of deleted records impossible.

Although prior work has focussed on designing *history independent* [1] data structures [5, 8, 12, 13, 16, 19], many challenges remain un-addressed for their successful deployment in systems. The first challenge is the un-availability of system-wide, space-allocation mechanisms that are also history independent. Such mechanisms are needed because data structures typically reside in storage sub-systems such as memory or disk. Hence, even if a data structure is history independent the behavior of the underlying storage sub-system may compromise its history. For example, the allocation of disk blocks by the file system can reveal information about the past operations performed on an otherwise history independent data structure (detailed examples in Section 3). Therefore, to realize complete history independence all system components on the data path need to possess the same characteristics [11]. Next, individual component characteristics render the existence of a single history independent design for all components unlikely. For instance, the low RAM latency vs. disk seek times, and the linear-ordered storage of disks vs. the wear leveling of SSDs. Hence, specific history independent designs are needed for each system component. Finally, achieving history independence efficiently in actual systems is hard due to the fact that current system designs heavily benefit from (data and time) locality at all layers through heavy caching, and existing history independent data structures completely destroy locality.

In this work we address the challenges of providing history independence for file storage over disk devices (Figure

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516724.

[1]A data structure is history independent if its storage layout is a function of the current state and not of the history of past operations that led to it.

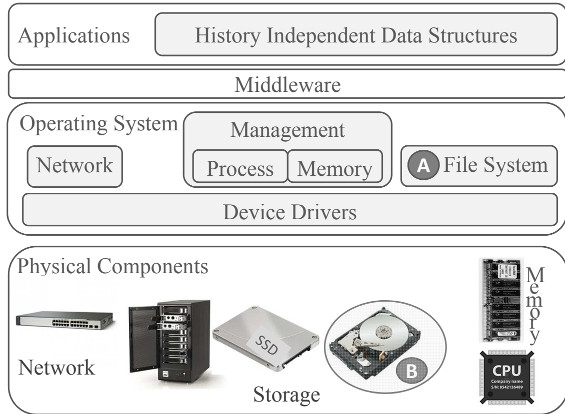


Figure 1: Potential system components that require history independent designs. This work targets file systems (A) using hard disks (B) for file storage.

1). We achieve this by introducing the first history independent file system (HIFS) with the following key design and performance characteristics.

(1) Varying degrees of data locality with minimal modifications for different application scenarios (Section 4.5.1).

(2) Customizable history independent layouts via simple modifications of a few key procedures (Section 4.5.1).

(3) Sequential read throughputs within 0.7x - 0.5x as compared to existing non-history independent file systems such as Ext3 for loads up to 60% (Section 6). Random read throughputs within 0.7x - 0.5x of Ext3 for loads up to 90%. Low performance for write operations at loads >60%.

(4) Efficient history independent file meta-data operations (Section 6). File delete and move operations as expensive as an entire file write (Section 4.5.4).

2. MODEL

Adversary. We assume an adversary with full access to the storage medium (e.g., the system disk). By forensic analysis of the disk contents the adversary aims to illegitimately derive sensitive information. Adversary actions include (but are not limited to) the following.

(1) Determine the existence and content of records deleted in the past, thereby violating regulatory compliance [4].

(2) Determine the order of past file operations to subvert privacy in voting applications [5].

(3) Compromise history independence of data structures stored within files via file system meta-data and disk layouts.

Storage Medium. The underlying storage device is required to be a mechanical disk drive, not flash storage (discussion on SSD storage in Section 5).

Files. All data structures stored within files are history independent [5, 8, 12, 13, 16, 19].

3. HISTORY INDEPENDENCE

3.1 File Systems

Existing file systems do not preserve history independence because the disk layouts they produce are not just a function of file contents but also depend on the sequence of file operations. The exact same set of files can be organized differently on disk depending on the sequence of operations that created the set. As a result, a simple examination of the disk layouts can reveal sensitive information about past operations. Moreover, the data structures used to maintain file system meta-data also contain information about past operations, both in their layouts, and in their contents (e.g., list of allocated blocks). Therefore, when disk layouts are combined with file system meta-data and knowledge of data structures that are stored within files, significantly more information can be derived, even full recovery of deleted data. Such leakage can violate secure deletion requirements and compromise privacy in applications such as e-voting.

At first it may appear that a simple replacement of all file system data structures (file meta-data, free block list etc) with their history independent versions would suffice to resolve the issue and preserve privacy. However, this is not the case, for the privacy leaks are not due to the data itself but due to the organization of data on disk. File system encryption too cannot solve the problem since the concern here is not data confidentiality. The relevant adversary is in fact an insider that requires and is rightfully granted full access to the data in the future (including disk encryption keys etc) and thus can easily bypass any protections encryption may offer. The solution then is to make file systems completely history independent. The disk layouts of a history independent file system (defined in Section 3.1.2) are only a function of data and not of the sequence of past operations.

3.1.1 Example Illustration

To illustrate the need for history independence in file systems we consider an admissions management application at a hospital that records patients' data as new patients are admitted. The application API permits the hospital staff to add new patient records, lookup existing records and delete patient records on discharge. We will use this example for illustration but note that in essence, most existing applications of history independent data structures cannot be securely realized in practice without an underlying file system providing history independent persistence.

The patient application will typically utilize a database to manage its data. Now, a database in turn stores and manipulates data by utilizing efficient data structures of which B-Trees [7] are the most common example. However, the use of B-Trees causes several privacy concerns. This is because the storage layout of B-Trees (or of variations such as B^+-Trees) often depends on the order in which operations are performed on them due to their deterministic insertions and deletions. For example, consider the the illustration in Figure 2 which shows two B-Trees that store the exact same elements. Yet the layouts of the two B-Trees differ due to different insertion orders. Hence, simply by examining the tree layout (Figure 2(a)) one can ascertain with a probability of 75% that *Chad* was admitted before *John*. This is due to the fact that out of the total 4! ways of insertion

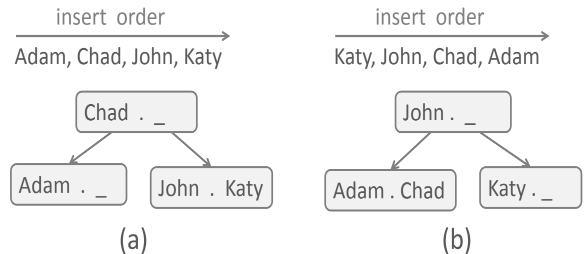


Figure 2: B-Tree, a non history independent data structure.

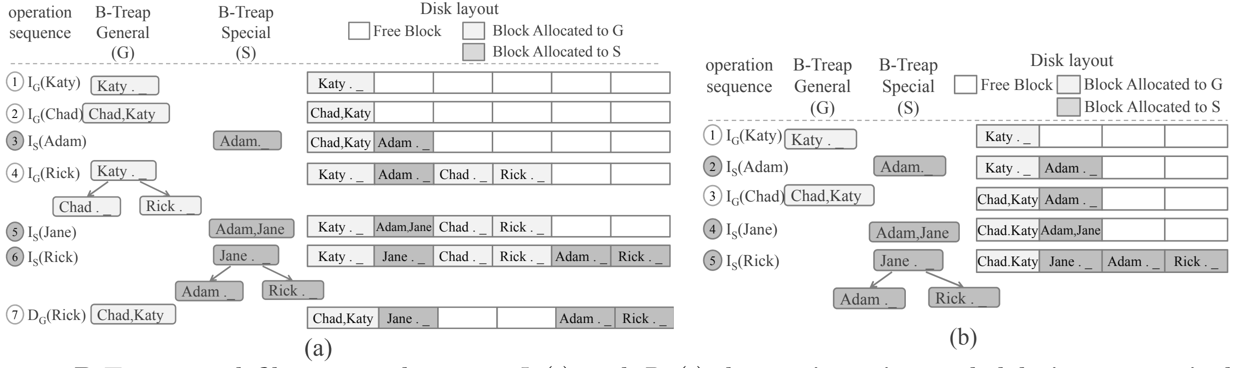


Figure 3: B-Treaps and file system layouts. $I_R(t)$ and $D_R(t)$ denote insertion and deletion respectively of element t in relation R.

for the four elements *Adam, Chad, John and Katy*, *Chad* is the root in only twelve, and inserted before *John* in nine of those twelve sequences.

The above limitation of B-Trees can be prevented by deploying corresponding history independent versions, such as B-Treaps [12]. The treaps will yield the same layout irrespective of the insertion order. However, such a simple replacement of application data structures with their history independent versions does not suffice unless the underlying persistent mechanisms (e.g., file system) are also history independent. To clarify, suppose that the B-Trees from above are replaced by the history independent B-Treaps [12]. Also, suppose that the application has two relations, one for admissions to the General ward and another for admissions to the Special ward, and that both relations are persisted using a simple file system that allocates the first available free block on request (and is hence not history independent). For simplicity, assume that the B-Treap node size is equal to the file system block size.

Now, consider the sequence of operations, the resultant B-Treaps, and the space allocation by the underlying file system as shown in Figure 3(a). At the end of operation 7, the disk layout (Figure 3(a)) reveals the following. (i) The fact that a delete operation was performed. The gaps left by the file system allocation form evidence for a delete. (ii) That the deleted node belonged to the General relation. Since otherwise there would be no gaps in the file system. (iii) The first patient was not admitted to the Special ward, since the root node of Special relation is not the first block on storage. Note that neither the layout of the B-Treaps nor the application API reveal any of (i) - (iii). The leaks are solely due to file system allocation.

Figure 3(b) shows an alternate sequence of operations that yield the exact same trees as in Figure 3(a) yet with significantly different disk layouts, showing that the disk layouts produced by the file system allocation heavily depend on file system operation sequencing. Hence, underlying storage mechanisms can defeat the history independence of higher level data structures.

While we used a simple file system scenario for illustration, we note that existing file systems (e.g., Ext2/Ext3 [3]) suffer from the same problems, since, for efficiency, and to preserve locality, they too allocate new blocks based on existing state, resulting in heavily history dependent layouts. A history independent file system on the other hand would reveal none of (i) to (iii) above since it would carry no evidence of a delete, such as gaps in block allocation. Similarly, its allocation policy would not be dependent on the order of file operations, resulting in the exact same disk layouts for the operation sequences of both Figures 3(a) and 3(b).

3.1.2 History Independent File System

Let Ψ^{Γ}_{F} denote the distribution (layout) over secondary storage of the state F, of a file system Γ. F represents all file contents plus meta-data for individual files and for the overall file system. Also, let $S^{\Gamma}_{F\rightarrow F'}$ denote a set of operations performed by file system Γ transforming its state from F into F'. We can then define the following.

DEFINITION 1. History Independent File System (HIFS) Γ *is a history independent file system if any two sequence of operations* $X^{\Gamma}_{F\rightarrow F'}$ *and* $Y^{\Gamma}_{F\rightarrow F'}$ *result in an identical distribution* $\Psi^{\Gamma}_{F'}$.

Formulated differently, for two sequences of operations $X^{\Gamma}_{F\rightarrow F'}$ and $Y^{\Gamma}_{F\rightarrow F'}$, an adversary having access to both the start state F and the end state F', should not be able to deduce which of X or Y was executed to transform the state from F into F'. It can be easily seen that satisfying this definition immediately implies that, the data at a particular offset of any file f in F, is always stored at the same device-specific location, irrespective of the sequence of operations that result in F. That is, the representation of the state F over the storage medium is canonical.

3.2 System-Wide History Independence

Although detailed discussion is out of scope, we briefly generalize history independence for an entire system.

Consider a typical system composed of multiple layers, ranging from software applications to hardware (Figure 1). Let these layers be denoted by L^0, L^1, L^2 etc, where L^0 is the bottom layer, L^1 the next higher layer and so on. Also, let $S^{L^i}_{A\rightarrow B}$ denote the set of operations that transform L^i from state A to B. Further suppose that, for each state A of L^i, there exists a state A' of L^{i-1} with the requirement that when L^i is in state A, L^{i-1} must be in state A' – we denote this relationship between A and A' as $L^i_A \rightarrow L^{i-1}_{A'}$. Then we can define the following.

DEFINITION 2. History Independent System
A system is history independent if (i) each system layer L^i *is history independent,* $i \geq 0$*, and (ii) any sequence of operations* $X^{L^i}_{A\rightarrow B}$ *causes* L^{i-1} *to change from state* A' *to* B'*, where* $L^i_A \rightarrow L^{i-1}_{A'}$, $L^i_B \rightarrow L^{i-1}_{B'}$ *and* $i > 0$.

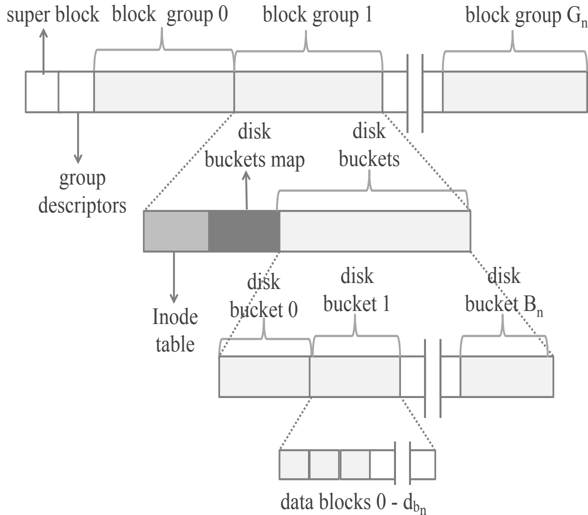


Figure 4: *HIFS* disk layout. Key parameters: $G_n \leftarrow$ number of block groups, $B_n \leftarrow$ number of disk buckets per block group, $d_s \leftarrow$ Data block size in bytes, $d_{b_n} \leftarrow$ number of data blocks per disk bucket.

4. ARCHITECTURE

4.1 Overview

The goals of the *HIFS* design are three-fold. (a) For any given set of files, the layout of their contents (including file and system meta-data) on disk are in a canonical form which is independent of the sequence of file operations, thus being history independent as per the definition 1. (b) Despite history independent layouts on disk, data locality is preserved. (c) The canonical layouts are customizable to suit a wide range of application requirements.

HIFS closely resembles existing Linux file systems such as Ext2 [3], exposing the exact same API and utilizing a similar disk structure (Figure 4). The key differences that give it history independent characteristics are the use of new locality-preserving history independent data structures for all file system meta-data (e.g., the inode table, Section 4.5.3) and the fact that the allocation of free disk blocks to files is not based on history. Hence *HIFS* does not use indirect and double indirect blocks to map file blocks to disk blocks. Instead, the entire data blocks section on disk is managed as a history independent data structure to allocate blocks to files (Section 4.5.1).

In the following sections we detail. Space constraints prevent too much in-depth detail on each operation. Instead we focus more on the features that specifically give HIFS its history independent characteristics.

4.2 History Independent Hash Table [5]

The key feature of *HIFS* is the replacement of all file system disk structures with history independent versions that we then endow with data locality preservation properties. The data structure of choice here is the history independent hash table in [5]. Hence, first we describe the hash table construction and in subsequent sections illustrate its use in various *HIFS* components.

The hash table in [5] is based on the stable matching property of the *Gale-Shapley Stable Marriage* algorithm [10] detailed in the following.

Stable Marriage Algorithm. Let M and W be a set of men and women respectively, $|M| = |W| = n$. Also, let each man in M rank all women in W as per his set of preferences. Similarly, each women in W ranks all men in M.

The goal of the stable marriage algorithm is to create n matchings (m, w) where $m \in M$ and $w \in W$ s.t no two pairs (m_i, w_j), (m_k, w_l) $(i \neq k, j \neq l)$ exist where (a) m_i ranks w_l higher than w_j and (b) w_l ranks m_i higher than m_k. If no such pairings exists then all matchings are considered stable.

The algorithm works as follows. In each round, a man m proposes to one woman at a time based on his ranking of W. If a woman w being proposed to is un-matched then a new match (m, w) is created. If the woman w is already matched to some other man m' then one of the following occurs. (a) if w ranks m higher than m' then the match (m', w) is broken and a new match (m, w) is created, or (b) if w ranks m lower than m', then m proposes to the next woman based on his rankings. The algorithm terminates when all men are matched.

[10] shows that if all the men propose in decreasing order of their preferences (ranks) then the resulting stable matching is unique. This holds even if the selection of a man m (who gets to propose) in each round is arbitrary.

History Independent Hash Table. [5] then uses the above property of the Stable Marriage algorithm to construct a history independent hash table as follows. (1) The set of keys to be inserted in to the table are considered as the set of men. (2) The set of hash table buckets are considered as the set of women. (3) Each key has an ordered preference of buckets and vice versa. (4) The preference order of each key is the order in which the buckets are probed for insertion, deletion and search. (5) In case of a collision between two keys, the key which ranks higher on the bucket's preference takes the slot. The lower ranked key is relocated to the next bucket in its preference list.

(1) - (5) ensure that the layout of keys in the hash table is the same irrespective of the sequence of key insertions and deletions, thereby making it history independent [5].

4.3 Key Insights

A simple replacement of all file system structures with the above history independent hash table will suffice to yield a history independent layout of files on disk. However, this neither preserves data locality nor gives the flexibility to choose different layouts based on application characteristics, both of which are key goals in the design of *HIFS*. Then a key observation in this context is the following. In the Stable Marriage algorithm each man in M can rank the n women in W in $n!$ ways, and vice-versa. Hence, several set of preferences from keys to buckets and buckets to keys are possible, each resulting in a distinct hash table instance. Therefore, by changing the preference order of keys and buckets we can control the layout of keys within the hash table.

The re-ordering of preferences leads to the realization that we can rewrite the algorithms in [5] to enable easy-custom selection of history independent layouts with minimal modifications. For this, we categorize the hash table operations in two Procedure Sets, a *generic* set and a *customizable* set. The generic procedures implement the overall search, insert and delete operations, and can be used unaltered for all scenarios. The customizable procedures determine the specific key and bucket preferences thereby governing the resultant hash table layouts.

Procedure Set 1 History Independent Hash Table

Procedure: INSERT
Desc: insert the given key in to the hash table.
Input: Tables $H^{0-m}[n]$, key k

1: $<i,r> \leftarrow$ GET_MOST_PREFERRED_BUCKET(k)
2: $c \leftarrow 0$
3: **while** $c < (n*(m+1))$ **do**
4:   **if** $H^r[i]$ is null **then**
5:     $H^r[i] \leftarrow k$
6:     **return** $<i,r>$
7:   **if** BUCKET_PREFERS($i, r, k, H^r[i]$) **then**
8:     SWAP($k, H^r[i]$)
9:   $<i,r> \leftarrow$ GET_NEXT_BUCKET(k, i, r)
10:   $c \leftarrow c+1$
11: **return** $<null, null>$ {tables are full}

Procedure: SEARCH
Desc: search for the given key in the hash table.
Input: Tables $H^{0-m}[n]$, key k

1: $<i,r> \leftarrow$ GET_MOST_PREFERRED_BUCKET(k)
2: $c \leftarrow 0$
3: **while** $c < (n*(m+1))$ AND $H^r[i]$ is not null **do**
4:   **if** k == $H^r[i]$ **then**
5:     **return** $<i,r>$ {key found at $H^r[i]$}
6:   $<i,r> \leftarrow$ GET_NEXT_BUCKET(k, i, r)
7:   $c \leftarrow c+1$
8: **return** $<null, null>$ {key not found}

Procedure: DELETE
Desc: delete the given key from the hash table.
Input: Tables $H^{0-m}[n]$, key k

1: $<i,r> \leftarrow$ SEARCH(k)
2: **while** i is not null AND $H^r[i]$ is not null **do**
3:   $<j,s> \leftarrow$ GET_NEXT_BUCKET(k, i, r)
4:   **if** $H^s[j]$ is not null AND KEY_PREFERS($H^s[j], i, j, r, s$) **then**
5:     $H^r[i] \leftarrow H^s[j]$, $k \leftarrow H^s[j]$, $i \leftarrow j, r \leftarrow s$

The generic procedures include INSERT, SEARCH and DELETE, listed in Procedure Set 1. These in turn use the customizable procedures, which include GET_MOST_ PREFERRED_BUCKET, GET_NEXT_BUCKET, BUCKET_PR EFERS and KEY_PREFERS. We list the scenario specific customizable procedures later in Sections 4.5.1 and 4.5.2 while discussing file system operations.

In short, this new procedure classification and rewrite enables new distinct history independent layouts for the same data set through modifications to the customizable procedures. Moreover, these modifications can now be targeted to maximize locality or to favor other application characteristics, e.g., read-only, sequential access etc. (Section 4.5.1).

4.4 Disk Layout

Not unlike Ext2 [3] *HIFS* divides the disk into block groups, each with its own inode table, map and data blocks. The super block contains information about the overall file system (e.g., number of block groups, disk block size etc.), and the individual group descriptors describe their respective block groups (e.g., inode table size, location of the disk buckets map etc.). Parameters affecting the disk layout (Figure 4) can be set up at file system creation time. The superblock and group descriptors have fixed sizes and occupy the same locations on disk independent of file contents. Hence, no special history independent designs are needed for them. The

Procedure Set 2 Customizable Procedures for Case A (Block Group Locality) from Section 4.5.1

Procedure: GDB
Desc: get the logical file bucket number from file offset.
Input: file offset f_o : $f_o \in \mathbb{N}$

1: **return** $\left(\left\lfloor \left\lfloor \frac{f_o}{d_s} \right\rfloor / d_{b_n} \right\rfloor\right)$

Procedure: GET_MOST_PREFERRED_BUCKET
Input: key k : k = {file path f_p, file offset f_o}

1: **return** $<h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n},\ h(f_p) \bmod \mathsf{G_n}>$

Procedure: GET_NEXT_BUCKET
Input: key k : $\{f_p, f_o\}$, bucket i, block group r : $(i, r) \in \mathbb{N}$

1: $i \leftarrow (i+1) \bmod \mathsf{B_n}$
2: **if** $i == (h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n})$ **then**
3:   $r \leftarrow (r+1) \bmod \mathsf{G_n}$, $i \leftarrow h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n}$
4: **return** $<i,r>$

Procedure: BUCKET_PREFERS
Input: bucket i, block group r : (i,r) $\in \mathbb{N}$, key a : $\{f_{p_a}, f_{o_a}\}$, key b : $\{f_{p_b}, f_{o_b}\}$

1: **return** $h(f_{p_a}||\text{GDB}(f_{o_a})) > h(f_{p_b}||\text{GDB}(f_{o_b}))$

Procedure: KEY_PREFERS
Input: key k : $\{f_p, f_o\}$, bucket i, bucket j, block group r, block group s : (i,j,r,s) $\in \mathbb{N}$

1: **if** r <>s **then**
2:   **return** $((h(f_p) \bmod \mathsf{G_n})$ - r + $\mathsf{G_n}) \bmod \mathsf{G_n} < ((h(f_p) \bmod \mathsf{G_n})$ - s + $\mathsf{G_n}) \bmod \mathsf{G_n}$
3: **return** $((h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n})$ - i + $\mathsf{B_n}) \bmod \mathsf{B_n} < ((h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n})$ - j + $\mathsf{B_n}) \bmod \mathsf{B_n}$

disk bucket maps and the inode tables however, play a critical role in history independent file storage, and we discuss them in detail below.

4.5 File Storage

Disk Buckets. File data is stored in blocks on disk. These are grouped into units, where each unit consists of a fixed number of (multiple) data blocks. Each such unit is termed as a disk bucket (Figure 4). Although read and write operations access individual data blocks, space is allocated to files in multiples of disk buckets.

Disk Buckets Map. *HIFS* relies on a special region in each block group referred to as the disk buckets map for allocation of new disk buckets to files and for locating disk blocks in read and write operations. Each entry within the disk buckets map has a one-one mapping to the corresponding disk bucket within that block group (Figure 4). The entry in the map contains meta-data about the corresponding disk bucket (e.g., whether the bucket is free or occupied).

All file system operations first locate the target entry in the disk buckets map and only then perform the actual read or write operation on the corresponding disk bucket. This avoids the need to perform expensive seek operations on actual file contents to locate data blocks. Also, the relative smaller size of the maps means that they are often cached in memory for faster access.

4.5.1 History Independent Layouts

Existing file systems such as Ext2 [3] maintain a list of allocated blocks within the file inode which renders the disk space allocation history dependent. *HIFS* however, does not rely on such lists; instead location of data blocks are

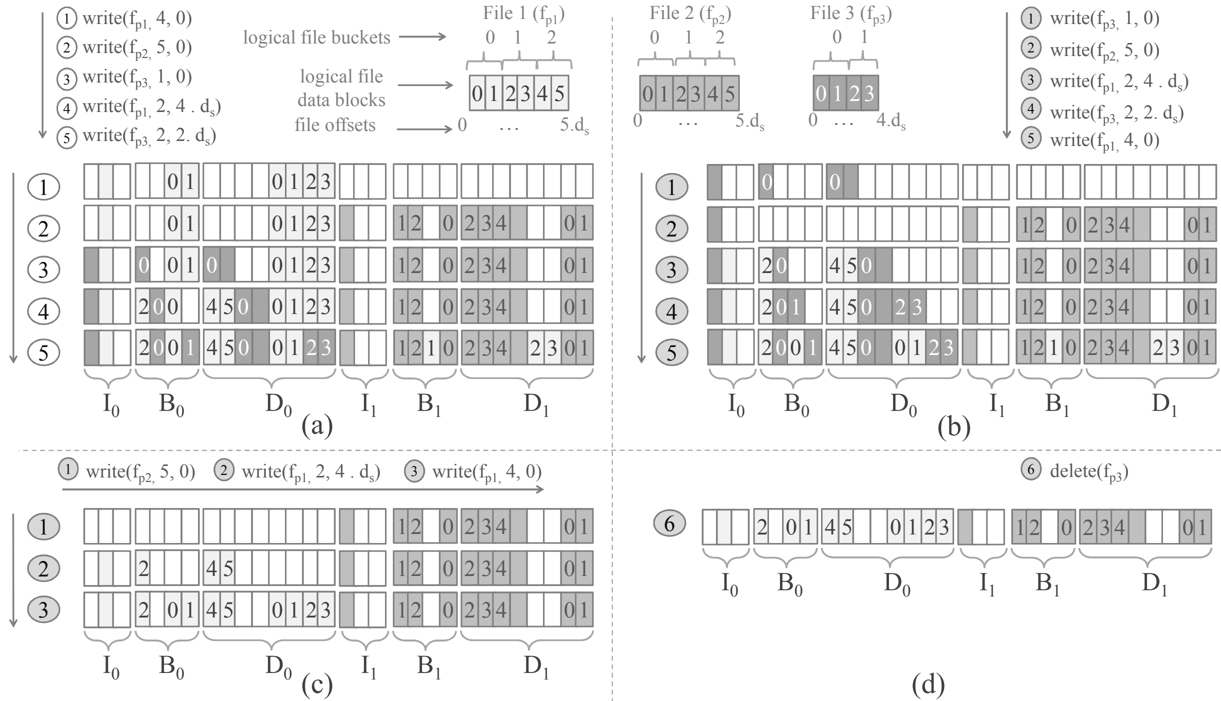


Figure 5: Sample executions and corresponding disk layouts for Case A from Section 4.5.1. Here $\mathsf{G_n} = 2$, $\mathsf{B_n} = 4$, $\mathsf{d_{b_n}} = 2$. $\mathsf{I_i}$, $\mathsf{B_i}$ and $\mathsf{D_i}$ denote the inode table, disk buckets map, and the disk blocks respectively of block group i. Also, $h(f_{p_1}) = 2$, $h(f_{p_2}) = 3$ and $h(f_{p_3}) = 4$. $write(f_p, f_d, f_o)$ represents a write operation of f_d blocks on file f_p at offset f_o. The disk blocks occupied by files with paths f_{p1}, f_{p2} and f_{p3} are shaded with their respective colors. Blocks that are not shaded indicate free blocks. The number within a block represents the data block number of the corresponding file.

derived directly from file attributes. Thus, for each read or write operation the location of data blocks on disk are determined only by the parameters to the current operation and do not depend on any past operation. To this end, the disk bucket maps from all block groups are collectively treated as a single history independent hash table. Hash table keys are derived from file attributes such as the file paths and read-write offsets. The entries in the maps are themselves the hash table buckets.

As discussed in Section 4.3, by altering the derivation of keys and the keys↔buckets preference sets we can attain customized layouts of the hash table to suit different requirements. Hence, file system operations use the generic hash table procedures from Procedure Set 1 to locate free blocks for allocation or to find existing blocks for reading or writing. Different history independent layouts of file contents on disk are realized solely by altering the customizable Procedure Set. Thus, the file system operations (read, write, mkdir etc) have implementations independent of the underlying history independent layouts, which in turn can be designed as needed, by specific customizable procedure implementations. To illustrate this, the actual file system write operation is included in Procedure Set 6 (Appendix).

We now describe a specific history independent layout scenario in detail and then briefly discuss others.

Case A: Block Group Locality. For data locality and overall efficiency, it is highly desirable that data blocks of the same file are located close together on disk, ideally within the same block group. To realize this scenario we tailor the customizable procedures as shown in Procedure Set 2. All other generic history independent hash table procedures from set 1, and the generic file system operations are unchanged.

To understand this better, consider Figure 5 which gives a detailed example for a set of three files. Each of Figures 5(a)-5(d) list a sequence of file system operations and depict the resultant disk layout (including files meta-data and content) at the end of each operation. Note that although the sequence of operations in Figures 5(a) and 5(b) differ, the resultant disk layouts at the end of either operation sequence are exactly the same (i.e., canonical).

To achieve canonical disk layouts the file system operations are translated in to hash table operations as follows: (a) Keys are derived from the full file path f_p and the read or write offset f_o. (b) The hash table buckets are the disk buckets map entries. (c) Key preferences are set such that each key first prefers all buckets from one specific block group, in a fixed order. Then, buckets from the next adjacent block group and so on. (d) Finally, buckets simply prefer keys with higher numerical values.

To give an intuition of how (a) - (d) preserve locality and yet give history independent layouts, consider the file system write operation (Procedure Set 6 in Appendix). The write operation first needs to locate the correct entry in the disk buckets maps. Once this entry is located it will perform the actual write on the corresponding disk bucket. Since the disk buckets maps from all block groups are treated as a single history independent hash table locating the correct buckets map entry is equivalent to finding the corresponding hash table bucket. Hence, locating the disk buckets map entry requires probing of the disk buckets maps. The probe order is exactly what is determined by the key preferences.

Locality. The probe order for any write operation always starts with the most preferred bucket as determined by the procedure GET_MOST_PREFERRED_BUCKET (Pro-

cedure Set 2). For a given file f_p, the block group of the most preferred bucket for all its write operations are derived only from the file path's hash $h(f_p)$ (line 1, second return parameter). Hence, all write operations of the same file begin their probing from the same block group independent of their target file offsets. Note that the probing for two write operations that target different offsets may start from two different buckets, but both buckets will be located within the same block group.

Moreover, subsequent buckets in the probing of the disk buckets map are determined using the GET_NEXT_BUCKET procedure which ensures that all buckets in the current selected block group are probed (line 1) before moving to the next adjacent block group (lines 2-4). Hence, data blocks of the same file prefer to be located in the same block group, preserving locality.

History Independence. For each bucket map entry in the probe sequence, the following two cases are possible.

(a) The bucket map entry is free. The data is written to the corresponding disk bucket and the key is stored in the bucket map entry, which is also marked as occupied.

(b) The bucket map entry is occupied by a previous write. A collision has occurred. Now it is exactly the collision resolution process that gives HIFS its history independence. To clarify, let w_1 and w_2 be two write operations on files f_{p_1} and f_{p_2} respectively. Also let w_1 target offset f_{o_1} of f_{p_1} and w_2 target offset f_{o_2} of f_{p_2}. Hence the keys for w_1 and w_2 for probing the disk bucket maps are $h(f_{p_1}||\text{GDB}(f_{o_1}))$ and $h(f_{p_2}||\text{GDB}(f_{o_2}))$ respectively (procedure GET_NEXT_BUCKET, line 2). Also, let $h(f_{p_1}||\text{GDB}(f_{o_1})) > h(f_{p_2}||\text{GDB}(f_{o_2}))$. Finally, suppose that the keys for both w_1 and w_2 prefer the same entry in the disk bucket map of the same block group and hence result in a collision. Now, there are two possible write sequences for w_1 and w_2. (i) *w_1 occurs before w_2:* Here, when w_2 is executing, the bucket is already occupied by the key of w_1. Also since $h(f_{p_1}||\text{GDB}(f_{o_1})) > h(f_{p_2}||\text{GDB}(f_{o_2}))$ the bucket prefers key of w_1 to that of w_2, as per procedure BUCKET_PREFERS (line 1). Hence, w_2 looks to the next bucket in its key's preference list and the bucket map entry remains unchanged. (ii) *w_2 occurs before w_1:* Here, the bucket entry is already occupied by the key of w_2 when w_1 is executing. But the bucket instead prefers the key of w_1 over that of w_2. Hence, the key of w_2 is now evicted from the bucket entry and is replaced by w_1's key. The key of w_2 is relocated to a new bucket based on its preference list and probe order. Also in this case, the data written by w_1 is now placed in the corresponding disk bucket, while the data previously written by w_2 is moved to the new disk bucket corresponding to the new bucket map entry determined for its key's relocation.

Figure 5 lists several examples of the two cases (i) and (ii) from above. Specifically, consider execution of operation 5 of Figure 5(b). Here, the write of data blocks zero and one of file f_{p_1} replaces and relocates the data blocks two and three of f_{p_3} which were previously written by operation 4.

As a result of (a) and (b), the bucket entry and hence the corresponding disk bucket contents are the same irrespective of the write sequence. Generalizing this example gives the following. For a given set of files F, and any operation sequence that creates F, a particular offset of a file f in F, is always stored at the same location on disk. That is, the disk layouts are history independent as per definition 1.

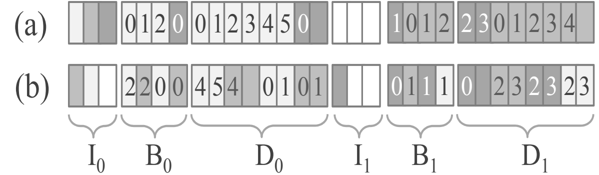


Figure 6: *HIFS* disk layouts for operation sequences of Figure 5. (a) Case B ← Complete Sequential and (b) Case C ← External Parameters, from Section 4.5.2. Also, $h(user(f_{p_1})) = 2$, $h(user(f_{p_2})) = 4$ and $h(user(f_{p_3})) = 3$.

4.5.2 *Customizing History Independence*

The above description (Case A) and the Procedure Set 2 provide just one specific scheme for a history independent disk layout for a given set of files. Different applications may prefer differing layouts based on their specific characteristics. Here we list additional examples that can be achieved using different implementations of the customizable procedures. The relevant customizable procedures for these scenarios are listed in Procedure Sets 3 and 4 (Appendix).

Case B: Completely Sequential. Applications that have very few writes as compared to read operations (e.g., databases for data mining) can greatly benefit if the entire file is stored sequentially on disk, giving maximal locality. To realize this, only the key and bucket preferences need to be modified. The keys of all write operations prefer buckets within the same block group as in case A above. However, the probe sequence starts with the first bucket in the block group and probes linearly henceforth. The buckets in turn prefer blocks in increasing order of file offsets.

Case C: External Parameters. Many applications access multiple files simultaneously [14]. I/O performance of such applications can be greatly enhanced if the files are located close together on disk. For example, files created by a single user/process are located in adjacency within the same block group. Again, this scenario can be realized with very minimal changes. In fact, the only change required is that key's preferences for block groups are now based on external (not file system related) parameters such as the user (procedure set 4, Appendix).

Figure 6 gives the resultant layouts for both cases B and C after the execution of operations from Figures 5(a) and 5(b). Again, note that the resultant layouts will be the same for both cases irrespective of the operation sequence.

Several other distributions are possible. Individual cases can also be combined to create more complex ones. For example, cases B and C can be combined to have a distribution wherein files from the same user are located adjacently on disk with each file laid out sequentially. Any new set of customizable procedures only needs to ensure that the key↔buckets preferences are unambiguous, that is, no key should prefer any two buckets equally and vice-versa. In Section 4.6, we show that as long as this condition is met the resulting distribution will be history independent.

4.5.3 *Inode Table*

Each inode is of fixed size and contains file meta-data such as file type, access rights, file size etc. Each inode table contains a fixed number of inode entries. Inodes are allocated to files at creation time.

Similar to the disk buckets maps, the inode tables from all block groups are collectively treated as a single history

independent hash table. Then, inode allocation and search is done using the generic procedures from Procedure Set 1. History independent layouts for the inode tables are also determined by the customizable procedures. One such set of inode-specific customizable procedures are listed in Procedure Set 5 (Appendix). Here the file inode is located in the same block group as that preferred by the file data blocks. The intuition and reasoning for history independence is similar to that described for case A above.

4.5.4 File delete, rename etc

HIFS also hides all evidence of a file delete operation. To illustrate, consider the sample executions and disk layouts from Figures 5(c) and 5(d). Figure 5(c) shows the resulting layouts of a sequence of write-only operations. Figure 5(d) shows the layout after execution of all operations of 5(b) plus the delete operation on file f_{p_3}. Both sequences create the exact same set of files. Note how the disk layout after the delete operation (Figure 5(d)) is exactly the same as the layout after the write-only sequence of Figure 5(c). This is because *HIFS* relocates data blocks on delete to their more preferred locations (procedure DELETE in Set 1). For example, in Figure 5(d) blocks 2 and 3 of file f_{p_1} are relocated to more preferred locations when file f_{p_3} is deleted (by operation 6). Overall, when a file is deleted, the resultant disk layout carries no traces of the delete operation, as if the file was never created in the first place.

The file delete is realized by execution of the delete procedure from Set 1 for each disk bucket allocated to the file. This makes a delete operation equivalent (cost-wise) to a write of the entire file. A rename or move operation for a particular file is a delete of each allocated disk bucket followed by a write (of the same bucket) with the new path.

4.6 Proofs of History Independence

Let K denote the set of keys and β denote the buckets within the hash table. Also, let $k.pref(b)$ denote bucket b's position on key k's preference list ($k \in K$ and $b \in \beta$) where, lower value of $k.pref$ indicates higher preference. Then, we have the following definition

DEFINITION 3. Un-ambiguous Preferences
A set of preferences from $K \rightarrow \beta$ *and* $\beta \rightarrow K$ *are* un-ambiguous *if*
(a) $\forall k \in K, \nexists (b_i, b_j) \in \beta, i \neq j$ *s.t.* $k.pref(b_i) = k.pref(b_j)$ *and*
(b) $\forall b \in \beta, \nexists (k_i, k_j) \in K, i \neq j$ *s.t.* $b.pref(k_i) = b.pref(k_j)$.

THEOREM 1. *The customizable procedures listed in Procedure Set 2 result in a history independent hash table with unique representation.*

PROOF. The Gale-Shapley Stable Marriage algorithm [10] has shown that if (a) men propose in decreasing order of their preference, and (b) all preferences are un-ambiguous, then the resulting stable matching is unique. [5] then showed that the hash table discussed in Section 4.2 satisfies both (a) and (b). They then proved that as a result the distribution of keys in the hash table is canonical and thus history independent. Thus, distributions from any set of customizable procedures are history independent if they exhibit the above two properties.

We now detail this for the customizable procedures in Procedure Set 2 applicable to case A (Block Group Locality) from Section 4.5.1.

(a) Proposal Order. Each time a new key is inserted in to the hash table, the first attempt is to place the key in its most preferred bucket i.e. bucket b where $k.pref(b)$ is minimum (see procedure INSERT in set 1 which foremost calls GET_MOST_PREFERRED_BUCKET in set 2). Each new bucket in the probe sequence is selected by the procedure GET_NEXT_BUCKET which given a bucket b_i finds the bucket b_j such that $\nexists\, b_l$ where $k.pref(b_i) < k.pref(b_l) < k.pref(b_j)$. In other words b_j is always the next preferred bucket on key k's preference list. Thus analogous to men in the stable marriage algorithm the matching of keys to buckets is attempted in decreasing order of key preferences.
(b) Un-ambiguous preferences. A bucket's preference between two keys is resolved in BUCKET_ PREFERS using the condition $h(f_{p_a}||\text{GDB}(f_{o_a})) > h(f_{p_b}||\text{GDB}(f_{o_b}))$. Hence, if the two values compared above are always distinct then the comparison is un-ambiguous. We note that for any given set of files F the combination of the file path and the logical file bucket number is unique, that is, $\forall f \in F$, $<f_p, \text{GDB}(f_o)>$is unique. Hence, the hash value $h(f_p||\text{GDB}(f_o))$ is unique. The proof then reduces to collision resistance of the hash function (e.g., SHA [9]).

Similarly, a key's preference amongst two buckets is resolved by the procedure KEY_PREFERS. Here, firstly if the two buckets belong to separate block groups (line 1) then the key simply prefers the bucket in the block group with the lower index (line 2). Hence, this condition is un-ambiguous. If the two buckets are in the same block group, then the key prefers the bucket closer to its most preferred bucket in that block group (line 3). Again, since the comparison is based on the hash of the unique combination $<f_p, \text{GDB}(f_o)>$the proof reduces to collision resistance of the hash function. □

Similar proofs exist for the customizable procedures of cases B and C from Section 4.5.2, and for the inode table.

5. DISCUSSION

Data Shredding. When an element is deleted from a history independent table it is imperative that proper data shredding is employed, that is, the data at the location is made irrecoverable (e.g., by overwriting [21]). Any residual data artifacts can compromise the history of operations. Hence, *HIFS* does not mark hash table entries as deleted but immediately performs an overwrite on each delete operation. This applies to all history independent disk structures (inode table, disk bucket maps) and to the file data blocks.
Temporal Meta-data. File systems typically maintain temporal meta-data such as modification times (for files and directories) which are then made available to applications. For certain composite applications it may be desired that the history of operations across files is also private. *HIFS* can optionally be configured to provide such functionality, which it does by (a) not maintaining any temporal meta-data for files, and (b) using history independent directories.

A directory file consists of a set of directory entries, one entry for each file in that directory. If cross-file history independence is desired then *HIFS* maintains the contents of each directory file in a history independent manner. For this, no additional data structures are employed, instead the directory entries are always stored sorted. Since a simple sorted list is history independent [20] the sorting of directory contents suffices.

System failure and Recovery. File system operations access multiple disk data structures which cannot all be updated simultaneously. Hence, in case of an inopportune system failure, the history independence related to certain records can be violated. For example, consider a file delete operation that accesses file system data structures in the following sequence – (a) deletes file inode entry from the inode table, (b) deletes an entry from the disk buckets map, and (c) deletes file contents from the corresponding disk bucket. Now, suppose that a system failure occurs after step (b) and an adversary gains access to the disk layout at this exact moment. By examining the disk buckets map and the data blocks on disk, the adversary can deduce that the last operation was a delete.

Existing recovery mechanisms such as journaling can be utilized to efficiently restore system consistency after a failure. However, we note that this breaks history independence since the adversary can potentially gain information about the recent operations. In case of journaling data can reside on disk without the associated updated meta-data (or vice versa). Hence data residing on disk with stale or missing meta-data is likely to be identified as recently written. Designing efficient recovery mechanisms that completely preserve history independence remains an open problem.

In-Memory History Independence. *HIFS* protects history independence only for data residing on disk. It does not target in-memory structures such as the file system cache.

Although the need for a history independent memory allocator for in-memory data structures has been voiced [11], we posit that extension of history independence to data in memory requires further careful examination. Simply replacing in-memory caches with history independent versions will not suffice. Firstly, an in-memory system cache cannot treat each disk block independently, but instead needs to maintain inter-block associations. For example, it must avoid scenarios such as the disk buckets map is written to disk but the modified file data blocks still reside in the cache for a significant time interval after. This can potentially reveal to an adversary details about the last file operation performed. Further, the relationships between in-memory data and its disk copies in a history independent context need to be investigated. This includes not just file system cache and data structures but also other system components such as the operating system kernel and low level system caches.

Storage Media. HIFS does not provide history independence if deployed (as is) on top of devices that manage their own internal block placement, such as SSDs, due to the wear-leveling mechanisms. The flash firmware's relocation of data blocks (wear leveling) is aimed to increase the life span of the drive. This essential functionality of the flash controller directly contradicts with history independence, since history independence requires fixed layouts. As of today we do not know how to reconcile both. Hence in HIFS we only focus on hard disk devices.

6. EXPERIMENTS

We benchmarked HIFS to understand the impact of history independence on performance.

Platform. All experiments were conducted on servers with 8 Intel i7 CPUs at 3.4GHz, 16GB RAM, and kernel v3.2.0-37. The storage devices of choice are Hitachi HDS72302 SCSI drives. The benchmark tool used is Filebench [1].

Parameter	**Database profile**	**Web Server profile**
File system size	100 GB	10 GB
Mean file size	1 GB	512 KB
No. of files	$L \cdot 100$	$(L \cdot 10) \cdot 2^{11}$
Disk block size ($\mathsf{d_s}$)	4 KB	4 KB
No. of block groups ($\mathsf{G_n}$)	8	24
Disk blocks / bucket ($\mathsf{d_{b_n}}$)	5120	128
Inode size	281 bytes	281 bytes
IO Size	32KB	512KB

Table 1: Experimental parameters. L $\leftarrow$ File System Load factor.

Implementation. *HIFS* is implemented as a C++ based user-space Fuse [2] file system. All data structures, including customizable history independence were written from scratch. The entire HIFS code is ≈10K LOC.

Measurements. Each test run commences with an empty file system, then creates and writes new files to storage. The number of files stored is increased until the file system is 90% full. Throughputs are measured at specific load factors (disk space utilization) ranging from 10% to 90%. The writes and subsequent read operations were separated by a complete clearing of the system cache. This was done to minimize the effect of caching since file system cache is not yet designed to be history independent.

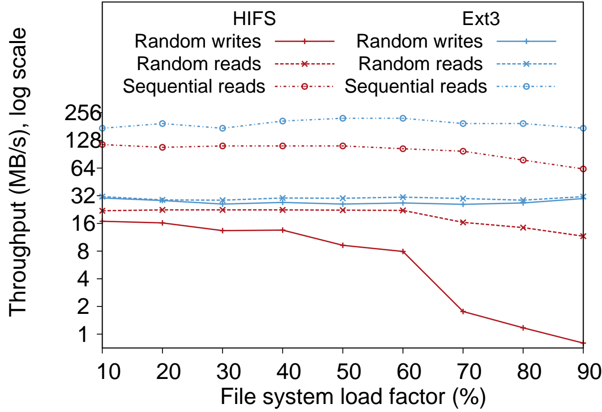
6.1 Results

Database Profile. Databases typically feature access to a few large files with random access patterns. To simulate such a profile we evaluate random reads and writes on multiple files with a mean file size of 1 GB. The number of files is varied from 10 to 90 to reflect the file system load factor. Table 1 summarizes all file system parameters while Figure 7(a) shows the results.

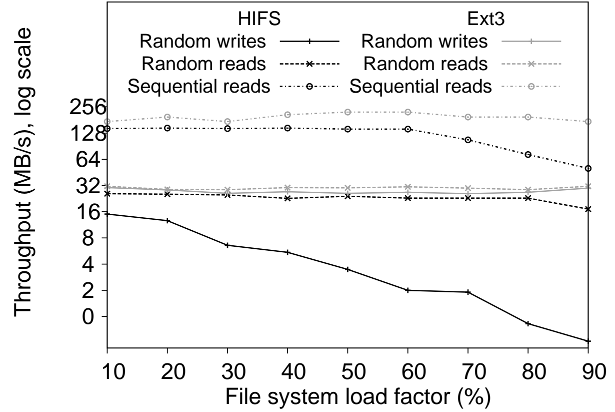
As detailed in Section 4.5.1, the allocation of a new disk bucket to a file in response to a write request potentially causes the displacement of existing data of other files due to the history independent collision resolution. The number of such collisions increases with the load factor. For load factors beyond 60% a sharp decrease in throughputs is observed for random writes. Read operations however, do not modify data, hence are not affected by collisions, in turn showing less throughput declines with load factors. Also, since the customizable procedures for case A (Block Group Locality, Section 4.5.1) are employed here, data locality is preserved. Write operations incur the overheads to maintain this locality while reads benefit from it.

The tradeoff between write and read performance is further evident from Figure 7(b) which shows the throughputs when the customizable procedures for case B (Complete Sequential, Section 4.5.1) are employed, that is, each file is laid out sequentially on disk. Here, read operations show a 13% average increase in throughput for sequential reads and 17% for random reads as compared to Case A (Figure 7(a)). To maintain this higher locality, write throughputs incur an average decline of 90% compared to Case A. All the above hold under history independence assurances.

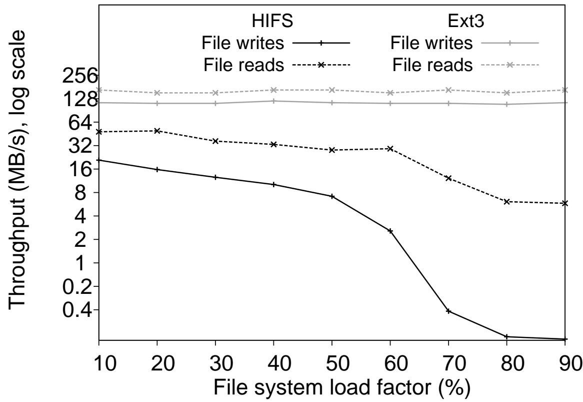
Web Server Profile. Web servers are characterized by accesses to a large number of very small files [22]. To model this, we use use a mean file size of 512 KB, but increase the number of files up to ≈18,500 for a load factor of 90%. Refer to Table 1 for a full parameter list. The read and



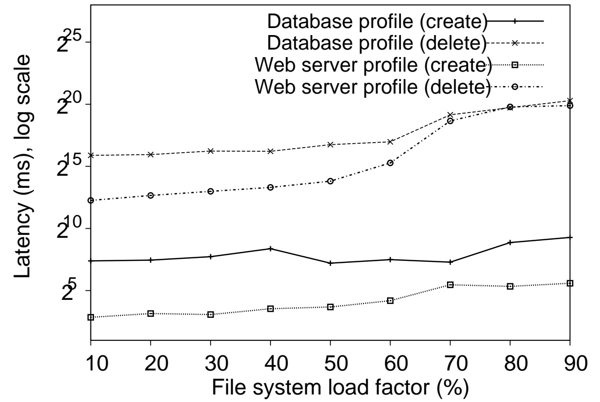
(a) Database profile, Case A: Block Group Locality (Section 4.5.1). HIFS in black, Ext3 in gray.



(b) Database profile, Case B: Complete Sequential (Section 4.5.1). HIFS in black, Ext3 in gray.



(c) Web Server profile, Case A: Block Group Locality (Section 4.5.1). HIFS in black, Ext3 in gray.



(d) HIFS latencies for file create and delete operations for Case A: Block Group Locality from Section 4.5.1.

Figure 7: HIFS throughputs and latencies. A load factor of L indicates that the file system is L% full.

write operations access entire files. Figure 7(c) summarizes the results for this profile.

File create and delete operations. Figure 7(d) shows the latencies of file create and delete operations.

A create operation involves locating a free inode for the new file, writing the file meta-data to the inode table, locating the parent directory and writing the file entry in to the directory. The writes to the inode table and to the directory are both history independent. Since the inode table also employs a history independent hash table to store individual inode entries (Section 4.5.3), the latency increases gradually with the load factor.

As discussed in Section 4.5.4 a file delete is equivalent to a write of the entire file in order to preserve history independence. As a result deletes have high latencies.

Overhead of history independence. Figures 7(a) and 7(c) also illustrate the throughputs for the Ext3 file system. For consistent comparisons all Ext3 file system operations are also routed via Fuse.

The performance of HIFS for read operations is comparable to read throughputs of Ext3 for load factors up to 70%. The write operations sustain significant overheads especially for higher load factors. This is again because once the write operations provide history independence while preserving locality, the reads incur significantly lower seek operations on the storage medium and are hence perform efficiently.

For the web server profile the overhead of history independence is higher even for reads. This is because each operation accesses the entire file at once and although the locality of data blocks withing a file is maintained the files themselves are distributed over the entire disk.

6.2 Summary and Analysis

HIFS employs history independent hash tables for all its data structures including file meta-data and data blocks. The performance of hash tables in turn depends on their load factor. In fact, the asymptotic performance per operation of the history independent hash table employed is $O(1/(1-\alpha)^3)$ [5], showing expected exponential decrease in performance with increasing load factor α. Hence, for load factors >60% the performance degrade for writes is significant. The same behavior is clearly evident in the experimental results for HIFS (Figures 7(a)-7(d)).

To remove these fundamental limitations on performance the following directions are most promising.

(1) Designing new history independent data structures specifically suited to secondary storage (current designs target only memory).

(2) Exploring trade-offs by lowering the degree of history independence, for example, batching of write operations. Delaying write operations will help reduce the total number of disk seeks and writes per operation.

Data Structure	Year	Ops	Runtime
2-3 Tree [16]	1997	I,L,D	$O(\log N)$
Hash Table [20]	2001	I,L	$O(log(1/(1-\alpha)))$
Hash Table [5]	2007	I,L,D	$O(1/(1-\alpha)^3)$
Ordered Dictionary [5]	2007	I,P,D	$O(\log\log N)$
Order Maintenance [5]	2007	I,C,D	$O(1)$
Hash Table [19]	2008	I,L,D	I,D $\rightarrow O(\log N)$ S $\rightarrow O(1)$
B-Treaps [12]	2009	I,D,R	$O(log_B N)$
B-SkipList [13]	2010	I,D,R	$O(log_B N)$
R-Trees [23]	2012	I,D,R	n/a

Table 2: Summary of history independent data structures. $\alpha \leftarrow$ load factor, $N \leftarrow$ number of keys, $B \leftarrow$ block transfer size. Also, I : insert, L : lookup, D : delete, R : range, P : predecessor, C : compare.

7. RELATED WORK

Prior work has focussed on designing several history independent data structures (summarized in Table 2). These data structures have several applications including incremental signature schemes [19], privacy in voting systems [5, 17–19], performing updates without revealing intermediate states [20], debugging parallel computations [5], reconciliation of dynamic sets [19], and un-traceable deletion [4].

Write Once Storage. The data structures in Table 2 assume a re-writable storage medium. [17] designed a history independent solution for a write-once medium suitable for deployment in voting machines. The construction is based on the observation from [20] that a lexicographic ordering of elements in a list is history independent. However, write-once memories do not allow in-place sorting of elements. Instead [17] employs copy-over lists [20]. When a new element is inserted, a new list is stored while the previous list is erased. However, this requires $O(n^2)$ space to store n keys.

[17] suggests to store each new element at a random location on the write-once storage. In case of collisions, a new random location is selected. Note that although simple and space-efficient, this requires the random bits to be hidden from the adversary which may not be possible in the targeted scenario involving voting machines in poll booths. [18] improves on [17] requiring only linear storage. The key idea here is to store all elements in a global hash table and for each entry of the hash table maintain a separate copy-over list which contains only the colliding elements.

Survey Works. Various definitions of history independence are analyzed in [15], most relying on canonical representations which are shown to be necessary in achieving it. [6] analyzes the lower and upper bounds on the runtime of the heap and the queue. A summary of various data structures from Table 2 is available in [11] along with techniques to transform basic data structures such as arrays, stacks and queues into their respective history independent versions.

8. CONCLUSION

The contributions of this paper are three-fold. First, we extend the concepts of history independence from in-memory data structures to file systems. Second, we devise a practical way to achieve history independence that preserves data locality. And last, we design, implement and evaluate the first history independent file system (HIFS). HIFS guarantees truly secure deletion by providing full history independence across both file system and disk layers of the storage stack. Additionally, it preserves data locality, and provides tunable efficiency knobs to suit different application history-sensitive scenarios.

9. REFERENCES

[1] Filebench, http://linux.die.net/man/1/filebench.

[2] Filesystem in userspace, http://fuse.sourceforge.net/.

[3] R. Appleton. Kernel korner: A non-technical look inside the ext2 file system. *Linux J.*, 1997(40es), Aug. 1997.

[4] S. Bajaj and R. Sion. Ficklebase: Looking into the future to erase the past. In *Proceedings of the International Conference on Data Engineering*, 2013.

[5] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 272–282. IEEE Computer Society, 2007.

[6] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history independence, 2003.

[7] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[8] S. A. Crosby and D. S. Wallach. Super-efficient aggregating history-independent persistent authenticated dictionaries. In *Proceedings of European conference on Research in computer security*, ESORICS, pages 671–688. Springer-Verlag, 2009.

[9] P. A. DesAutels. SHA1: Secure Hash Algorithm, www.w3.org/PICS/DSig/SHA1_1_0.html. 1997.

[10] D. Gale and L. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–15, 1962.

[11] D. Golovin. *Uniquely represented data structures with applications to privacy.* PhD thesis, 2008. AAI3340637.

[12] D. Golovin. B-treaps: A uniquely represented alternative to b-trees. In *Proceedings of International Colloquium on Automata, Languages and Programming: Part I*, ICALP '09, pages 487–499. Springer-Verlag, 2009.

[13] D. Golovin. The b-skip-list: A simpler uniquely represented alternative to b-trees. *CoRR*, abs/1005.0662, 2010.

[14] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. *ACM Trans. Comput. Syst.*, 30(3):10:1–10:39, Aug. 2012.

[15] J. Hartline, E. Hong, A. Mohr, E. E. Mohr, W. Pentney, and E. Rocke. Characterizing history independent data structures, 2002.

[16] D. Micciancio. An oblivious data structure and its applications to cryptography. In *In Proceedings of ACM Symposium on the Theory of Computing*, pages 456–464. ACM Press, 1997.

[17] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Tamper-evident, history-independent, subliminal-free data structures on prom storage-or-how to store ballots on a voting machine (extended abstract). In *Proceedings of IEEE Symposium on Security and Privacy*, SP '06, pages 365–370. IEEE Computer Society, 2006.

[18] T. Moran, M. Naor, and G. Segev. Deterministic history-independent strategies for storing information on write-once memories. In *Proceedings of International Colloquium on Automata, Languages and Programming*, pages 303–315. Springer, 2007.

[19] M. Naor, G. Segev, and U. Wieder. History-independent cuckoo hashing. In *Proceedings of international colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 631–642. Springer-Verlag, 2008.

[20] M. Naor and V. Teague. Anti-persistence: History independent data structures. In *In Proceedings of ACM symposium on Theory of computing*, pages 492–501. ACM Press, 2001.

[21] J. Reardon, D. Basin, and S. Capkun. Sok: Secure data deletion. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 301–315. IEEE Computer Society, 2013.

[22] A. S. Tanenbaum, J. N. Herder, and H. Bos. File size distribution on unix systems: then and now. *SIGOPS Oper. Syst. Rev.*, 40(1):100–104, Jan. 2006.

[23] T. Tzouramanis. History-independence: a fresh look at the case of r-trees. In *Proceedings of ACM Symposium on Applied Computing*, SAC '12, pages 7–12. ACM, 2012.

APPENDIX

A. PROCEDURE SETS

Procedure Set 3 Customizable Procedures for Case B (Complete Sequential) from Section 4.5.2

Procedure: GET_MOST_PREFERRED_BUCKET
Desc: get the top most bucket on key's preference list.
Input: key $k = \{$file path f_p, file offset $f_o\}$
1: **return** $<0, h(f_p) \bmod \mathsf{G_n}>$

Procedure: GET_NEXT_BUCKET
Desc: get the next bucket on key's preference list.
Input: key $k = \{f_p, f_o\}$, bucket i, block group r
1: $i \leftarrow (i+1) \bmod \mathsf{B_n}$
2: **if** $i == 0$ **then**
3: **return** $<0, (r+1) \bmod \mathsf{G_n}>$
4: **return** $<i, r>$

Procedure: BUCKET_PREFERS
Desc: find which of two keys the given bucket prefers.
Input: bucket i, block group r, key $a = \{f_{p_a}, f_{o_a}\}$, key $b = \{f_{p_b}, f_{o_b}\}$
1: **if** $f_{p_a} == f_{p_b}$ **then**
2: **return** GDB(f_{o_a}) <GDB(f_{o_b})
3: **return** $h(f_{p_a}) > h(f_{p_b})$

Procedure: KEY_PREFERS
Desc: find which of two buckets the given key prefers.
Input: key $k = \{f_p, f_o\}$, bucket i, bucket j, block group r, block group s
1: **if** $r \neq s$ **then**
2: **return** $((h(f_p) \bmod \mathsf{G_n}) - r + \mathsf{G_n}) \bmod \mathsf{G_n} < ((h(f_p) \bmod \mathsf{G_n}) - s + \mathsf{G_n}) \bmod \mathsf{G_n}$
3: **return** $i < j$

Procedure Set 4 Customizable Procedures for Case C (External parameters) from Section 4.5.2

Procedure: GET_MOST_PREFERRED_BUCKET
Input: key k : $k = \{$file path f_p, file offset f_o, user $u\}$
1: **return** $<h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n},\ h(u) \bmod \mathsf{G_n}>$

GET_NEXT_BUCKET and BUCKET_PREFERS same as in procedure set 2.

Procedure: KEY_PREFERS
Input: key k : $\{f_p, f_o, u\}$, bucket i, bucket j, block group r, block group s : $(i, j, r, s) \in \mathbb{N}$
1: **if** $r \neq s$ **then**
2: **return** $((h(u) \bmod \mathsf{G_n}) - r + \mathsf{G_n}) \bmod \mathsf{G_n} < ((h(u) mod \mathsf{G_n}) - s + \mathsf{G_n}) \bmod \mathsf{G_n}$
3: **return** $((h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n}) - i + \mathsf{B_n}) \bmod \mathsf{B_n} < ((h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n}) - j + \mathsf{B_n}) \bmod \mathsf{B_n}$

Procedure Set 5 Inode Table Customizable Procedures

Procedure: GET_MOST_PREFERRED_BUCKET
Input: key $k =$ file path f_p
1: **return** $<h(f_p) \bmod \mathsf{I_n},\ h(f_p) \bmod \mathsf{G_n}>$

Procedure: GET_NEXT_BUCKET
Input: key $k = f_p$, bucket i, block group r
1: $i \leftarrow (i+1) \bmod \mathsf{I_n}$
2: **if** $i == (h(f_p) \bmod \mathsf{I_n})$ **then**
3: $r \leftarrow (r+1) \bmod \mathsf{G_n}$, $i \leftarrow h(f_p) \bmod \mathsf{I_n}$
4: **return** $<i, r>$

Procedure: BUCKET_PREFERS
Input: bucket i, block group r, key $a = f_{p_a}$, key $b = f_{p_b}$
1: **return** $h(f_{p_a}) > h(f_{p_b})$

Procedure: KEY_PREFERS
Input: key $k = f_p$, bucket i, bucket j, block group r, block group s
1: **if** $r \neq s$ **then**
2: **return** $((h(f_p) \bmod \mathsf{G_n}) - r + \mathsf{G_n}) \bmod \mathsf{G_n} < ((h(f_p) \bmod \mathsf{G_n}) - s + \mathsf{G_n}) \bmod \mathsf{G_n}$
3: **return** $((h(f_p) \bmod \mathsf{I_n}) - i + \mathsf{I_n}) \bmod \mathsf{I_n} < ((h(f_p) \bmod \mathsf{I_n}) - j + \mathsf{I_n}) \bmod \mathsf{I_n}$

Procedure Set 6 *HIFS* write operation

Procedure: write
Input: file path f_p, file offset f_o, data ∂, data length ∂_l $\mathsf{I_t^{G_n}}$: Inode tables, $\mathsf{B_t^{G_n}}$: Disk Bucket Maps, $\mathsf{GD_t}$: Group Descriptor table
1: $<i_i, i_n> \leftarrow$ SEARCH($\mathsf{I_t^{G_n}}$, f_p)
2: check file permissions in inode data
3: $b_s \leftarrow (\mathsf{d_{b_n}} * \mathsf{d_s})$, $w_l \leftarrow 0$
4: **while** $w_l < \partial_l$ **do**
5: $<b_i, g_i> \leftarrow$ SEARCH($\mathsf{B_t^{G_n}}$, $\{f_p, f_o\}$)
6: **if** b_i is null **then**
7: $<b_i, g_i> \leftarrow$ INSERT($\mathsf{B_t^{G_n}}$, $\{f_p, f_o\}, \{\}$)
8: **if** $(\partial_l - w_l) > (b_s - (f_o \bmod b_s))$ **then**
9: $\partial_i \leftarrow b_s - (f_o \bmod b_s)$
10: **else**
11: $\partial_i \leftarrow \partial_l - w_l$
12: $s_d \leftarrow$ start offset of disk buckets in $\mathsf{GD}_t[g_i]$
13: write $\partial[w_l : w_l + \partial_i]$ bytes to disk at offset $s_d + (b_s * b_i) + (f_o \bmod b_s)$
14: $f_o \leftarrow f_o + \partial_i$, $w_l \leftarrow w_l + \partial_i$