



August 3, 2011

0.1 Introduction

1

Flash memory is an essential component of embedded devices. NOR flash is a kind of non-volatile memory that is compatible with the processor memory interface. As a result, its major application is the storage of binary executables. Because embedded devices have critical constraints on hardware costs and form factors, they often have no choice but to partition a piece of NOR flash into two regions: one for read-only executables and the other for read-write data. The read-write partition can adopt various index structures for fast data access. This design helps conserve the precious resources of CPU cycles and battery power.

NOR flash, as a kind of byte-addressable memory, can adopt various index structures developed for internal memory, such as balanced binary search trees [20]. However, the physical constraints of NOR flash make it difficult to implement these index structures in NOR flash. For example, erasing a memory location in NOR flash must precede writing to this memory location. Further, NOR flash typically erases in large blocks of 128 KB or more [15]. To avoid erasing NOR flash before every update, data can be updated out of place. However, out-of-place updates change the physical residence of data in NOR flash, leaving updated data pointers dangling. Fixing these dangling pointers updates them out of place as well. Updating a pointer also requires rewriting the data object possessing this pointer. However, this in turn produces more dangling pointers, recursively propagating out-of-place updates to a large number of data objects.

A common approach addressing this issue is to use logical pointers. Instead of using physical addresses in NOR flash, logical pointers refer to data objects in terms of logical addresses, which are uniquely assigned to every data object. This way, updating data objects out of place leaves their logical addresses intact, producing no logical pointers dangling. However, using logical pointers has two main drawbacks: First, translating logical addresses to physical addresses requires a RAM-resident mapping table. This mapping table can be considerably large, making it unaffordable for many small embedded devices. Second, this mapping table disappears when powering off, and may require scanning the entire flash memory to rebuild when powering on. This scanning process can be lengthy,

contradicting the requirement that embedded devices be instantly operational after powering on.

This study investigates efficient data indexing in NOR flash to address the drawbacks of using logical pointers and physical pointers. Specifically, this study is concerned with small embedded devices that use NOR flash have limited RAM and require a fast system boot. Many deeply embedded devices have these characteristics, including toys, facsimile machines, and wireless sensor nodes [8, 17].

This study proposes a native index structure for NOR flash, called *soft lists*. Soft lists organize data objects using *soft pointers*. Soft pointers use physical addresses, removing the need for a mapping table or boot-up scanning. Unlike physical pointers, the soft pointers approach allows pointer de-referencing to probe a bounded number of memory locations in NOR flash. This design has two advantages: First, it is possible to move data objects around in NOR flash without invalidating a soft pointer. This greatly simplifies data updating and space management in NOR flash. Second, a soft pointer is related to a number of data objects, including the random objects probed when de-referencing this soft pointer. When searching for a key in a soft list, judiciously following a probed random object can skip over a large number of objects, greatly speeding up the search.

Even though forward random skips greatly speed up search operations, searching in a soft list may degrade into a linear search in a worst case scenario. Therefore, this study proposes organizing multiple soft lists in parallel, producing a multilevel soft list. A multilevel soft list is structurally similar to a skip list [22]. Searching a key begins with the highest-level list. The search skips over a large number of keys in a few steps, and then descends to lower-level lists for short-range skips to locate the searched key. Multilevel soft lists do not require self-balancing, but rely on randomization, eliminating many expensive write operations to flash memory. For space management in NOR flash, this study introduces a space-allocation policy that jointly considers 1) the random nature of soft pointers, 2) the layered structure of multilevel soft lists, and 3) wear-leveling issues. The experiments presented in this study evaluate and compare soft lists against a tree-based index structure under synthesized workloads and real-life workloads. Results show that soft lists achieve significantly faster response times.

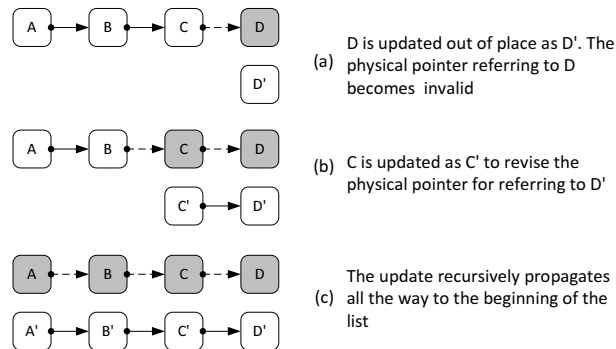


Figure 1: A scenario of how an update propagates to all the data objects in a list.

The rest of this paper is organized as follows: Section 2 introduces the background of this work. Section 3 presents the design of single-level soft lists, and Section 4 improves the scalability of this design using multilevel soft lists. Section 5 includes the experimental results, and Section 6 concludes this paper.

0.2 Background

0.2.1 Using Physical Pointers in NOR flash

Like physical pointers, soft pointers use physical addresses to address data objects. This section describes the main issues of using physical pointers in NOR flash.

NOR flash is a kind of erasure-based memory. To avoid erasing NOR flash every time a piece of data is updated, NOR flash data are updated out of place. Consider the example shown in Fig. 1: A list organizes four data objects using physical pointers. When updating D, new data D' is written out of place, leaving the physical pointer of C dangling. Since it is not possible to update the pointer of C in place, C must be rewritten out of place. As a result, the pointers and data objects all the way to the beginning of this list must be updated in turn. This issue is referred to as *pointer-update propagation*.

Pointer-update propagation can impose considerable overhead on the management of

index structures. Even worse, it can prevent index structures from completing index operations. Consider the example in Fig. 2(a), in which a block has four valid objects: A, B, C, and D. Suppose that this block is chosen as an erasure victim for free-space reclaiming. Before erasing this block, all the valid objects must be moved to a spare block, as Fig. 2(b) shows. This move leaves the physical pointers for objects a, b, c, and d dangling. Updating objects a, b, c, and d out of place can fix their dangling pointers. However, this invalidates more pointers, recursively triggering another batch of updates to fix the new dangling pointers. Because consuming an unbounded amount of free space precedes erasing a block for free-space reclaiming, the system can easily be deadlocked. This issue is *garbage-collection deadlock*.

Let f represent free space which is clean, d represent dead space which has been written whether valid or not, and $f + d$ is constant. Before free-space reclaiming, the distribution of free space and dead space is (f, d) . It is about to reclaim free space, so f is the minimum free space requirement for free-space reclaiming. After choosing an erasure victim, we could expect the distribution after free-space reclaiming will be (f', d') according to the number of valid objects of the chosen victim, where $f' > f$, and $d' < d$. Actually after free-space reclaiming, the distribution will be (f'', d'') , where $f'' < f'$, and $d'' > d'$. This is because of pointer-update propagation. When copying a valid object from victim to other free space, the other object point to the copied object must also be updated to free space. Therefore, after free-space reclaiming, the free space will less than we expect. In the worst case, $f'' < f$, and $d'' > d$. The free space after reclaiming may even less than the minimum free space requirement for free-space reclaiming. Hence, this situation would cause garbage-collection deadlock.

0.2.2 Related Work

Data indexing using flash memory is an increasingly important design issue for embedded software. Because NAND flash reads and writes in terms of 4KB pages [16], block-based index structures like B-trees are better suited to NAND flash. Random updates to B-trees partially modify tree nodes in NAND-flash pages, incurring considerable read-modify-write

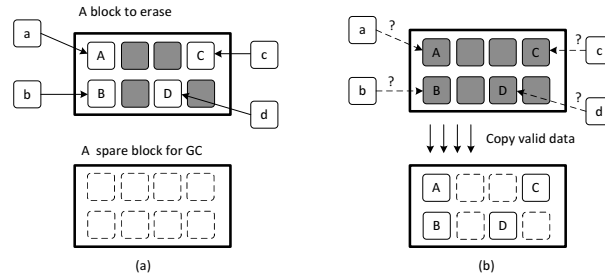


Figure 2: A problem caused by garbage collection. (a) A block has valid data A, B, C, and D. (b) Before erasing this block, garbage collection moves the valid data to a spare block, leaving many dangling pointers.

overhead and ancillary garbage-collection activities. To reduce the write traffic to NAND flash, Lee et al. [19], Li et al. [21], and Agrawal et al. [6] proposed various techniques that collect random B-tree updates in a sequential log associated with a high-level tree node, convert small and random updates into batches, and then cascade (i.e., copy) the update batches to lower-level node logs. This approach effectively reduces the read-modify-write traffic to NAND flash.

Lin et al. proposed MicroHash [17], which is a NAND-flash implementation of hash tables for sensor nodes. MicroHash treats flash memory as a circular log space. It appends new records to the log and discards old records from the beginning of the log. MicroHash is applicable only if there is a chronological order of all data and old data can be abandoned.

Prior studies also use B-trees on NAND-flash-based solid-state disks. Solid-state disks implement NAND-flash translation layer (i.e., NFTL [5, 14]) in firmware, hiding flash-memory geometry and management activities from the hosts of solid-state disks. Wu et al. [3] introduced a software layer between a standard B-tree implementation and the disk interface of solid-state disks. This layer logs index operations out of place, and then updates B-tree nodes in place at the proper time. Nath et al. [18] proposed adaptively switching a B-tree node between an in-place updating mode and a out-of-place logging mode. The LA-tree method proposed by Agrawal et al. [6] is essentially a B-tree partitioned into many sub-trees. Taking advantage of the temporal and spatial localities of accesses to B-trees, the LA-tree method limits logging and updating operations to each individual sub-tree.

Updating a data item out of place changes the data item's physical residence in flash

memory. To avoid producing dangling pointers, a common approach is to associate all data items with a unique logical address. In this case, a RAM-resident table is required to map logical addresses to physical addresses. Several research efforts have been aimed at reducing the size of this mapping table. Park et al. [5] and Lee et al. [14] proposed combining coarse-grained mapping and fine-grained mapping for address translation. Chang et al. [12] proposed a variable-granularity mapping scheme. Because the mapping table is stored in volatile memory, rebuilding this mapping table after power-offs may require scanning the entire flash memory. Wu et al. [4] and Yim et al. [11] proposed writing summary information in convenient flash-memory locations to speed up this scan procedure.

Many deeply-embedded devices are equipped with a small amount of RAM and a piece of NOR flash. Using logical pointers is infeasible in such devices because of the need for a large mapping table. A B⁺-tree variant proposed by Kang et al., called μ -trees [7], shares many design goals with this study. The μ -trees method uses physical pointers, eliminating the need for address translation. Modifying a leaf node in a μ -tree involves revising the physical pointers of the nodes all the way to the root node. Taking advantage of the fact that a page is the smallest read-write unit in NAND flash, μ -trees pack all the involved nodes in a page, and update these nodes out of place by one page write. However, unlike NAND flash, NOR flash is byte-addressable. Because a μ -tree updates many nodes for each update, using a μ -tree in NOR flash can incur an unacceptable write overhead, slowing down index operations.

0.3 Simple Soft Lists: The Basic Form

This section introduces the concept of soft pointers and the design of simple soft lists. Simple soft lists are single-level lists, which extends to multilevel soft lists for better scalability in the next section.

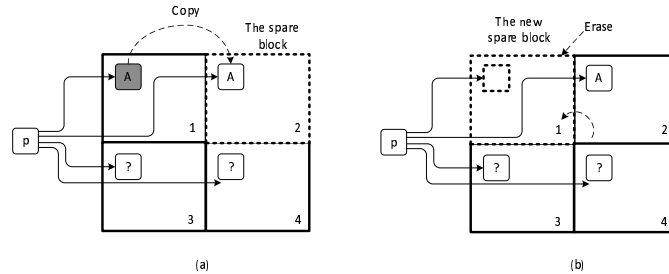


Figure 3: A turnstile consists of four blocks. Object “p” refers to object “A” using a soft pointer. (a) Garbage collection shifts object A in block 2 to the spare block, block 1. (b) Block 1 is erased into a spare block. Object p retains its reference to object A because de-referencing the soft pointer probes all objects having the same block offset in the turnstile.

0.3.1 Index Objects, Soft Pointers, and Turnstiles

An index object is the smallest element in soft lists, and consists of a key, a value, and a soft pointer. In the rest of this paper, we shall interchangeably use the terms index object, object, and key to refer to the same thing.

Like physical pointers, a *soft pointer* uses physical addresses to address index objects. However, de-referencing a soft pointer probes a number of memory locations, allowing a soft pointer to be related to many memory locations. If an index object is moved around in these pre-determined locations, then a soft pointer can never lose this index object.

This study implements soft pointers using *turnstile*s. The entire flash memory is partitioned into turnstiles, and each turnstile comprises a fixed number of flash blocks. Each turnstile reserves one block as spare space for garbage collection. Figure 3(a) shows a turnstile, which consists of four blocks, and block 2 is a spare block. Object p outside of the turnstile refers to object A in block 1 inside of the turnstile via a soft pointer. Besides object A, this soft pointer refers to all objects having the same offsets in other blocks, namely, the two unknown objects in block 3 and block 4. Now, let garbage collection select block 1 as a victim for erasure. As Fig. 3(b) shows, object A is “shifted” to block 2, the spare block, and block 1 is then erased into a spare block. Object A is still related to the soft pointer after garbage collection because its block offset does not change.

Conceptually, garbage collection rotates a turnstile counterclockwise, erasing all blocks

in the turnstile in a round-robin fashion. As a result, wear leveling of the blocks in a turnstile is perfect. However, if a victim block is far from the current spare block, then garbage collection must rotate the turnstile multiple times, erasing many blocks. Section 0.4.2 addresses this performance issue.

For ease of presentation, additional terms related to soft pointers are defined here: The *turnstile size* is how many blocks a turnstile has. The *soft-pointer degree* is how many non-spare blocks are in a turnstile. In Fig. 3, the turnstile size and the soft-pointer degree are four and three, respectively. The index object that a soft pointer refers to is the *target object* of this soft pointer. Except the target object, all the objects that can be probed during de-referencing a soft pointer are the *buddy objects* of this soft pointer. In Fig. 3(a), object A is the target object, while the two objects in block 3 block 4 are buddy objects.

0.3.2 Key Search

Figure 4(a) shows an ordinary linearly-ordered list using ordinary physical pointers. In a search session, the key to be located is the *searched key*, and the key currently visited is the *current key*. To search for a key, starting from the leftmost key in the list, the current key is iteratively moved toward the largest key, until the current key is no smaller than the searched key. At this time, if the current key matches the searched key, then this search is reported successful. Otherwise the searched key can not be found. In Fig. 4(a), searching key 200 visits 7 keys.

A simple soft list is structurally similar to a linearly ordered list. Replacing the physical pointers of the list in Fig. 4(a) with soft pointers produces a soft list depicted in Fig. 4(b). Every object has more than one outward edges, showing that a soft pointer relates an object to many objects. Figure 4(c) depicts a possible layout of this soft list in NOR flash. For example, key 10 refers to keys 15 and 55 via a soft pointer, because the latter two have the same block offset in the middle turnstile.

When searching for a key in a soft list, the current key moves to the right (i.e., forward) until *all* the probed keys are larger than the searched key or smaller than the current key.

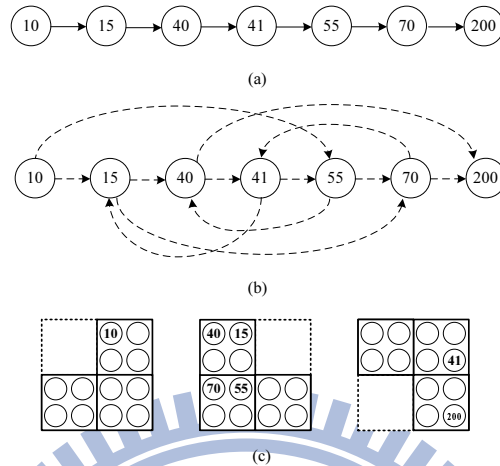


Figure 4: (a) A linearly ordered list. (b) A corresponding soft list with soft pointers. (c) Three turnstiles show a possible scenario of this soft list in NOR flash.

For example, to search key 200 in Fig. 4(b), beginning with key 10, the current key goes to keys 15, 70, and then 200. The current key skips forward to an object no matter what type the object is (i.e., a target object or a buddy object), and finishes in only four steps. Forward skips to buddy objects are referred to as *forward random skips* because the buddy objects are random objects. All the forward moves are greedy, so it is not necessary to differentiate a target object from buddy objects when de-referencing a soft pointer.

Now consider the case of searching a non-existing key 16. Starting from key 10, the current key goes to key 15. At key 15, the first probe gives key 40. Because key 40 is larger than 16, a second probe is necessary, giving key 70. As the two probed keys 40 and 70 are both larger than the searched key 16, it is reported that key 16 cannot be found.

De-referencing a soft pointer does not always produce useful probes. For example, consider a search for key 55. Beginning with the first key 10, the current key moves to 15 and then 40. At key 40, the first probe gives key 200, which is larger than the searched key 55. This probe is useless because it skips beyond the searched key. The current key falls back to key 40, and then takes the second probe and advances to key 41. At key 41, the first probe refers to key 15. This probe is useless because it causes a backward skip. At key 41, the second probe refers to key 55, so the searched key 55 is reported found.

Algorithm 1 shows the search algorithm of soft lists. Steps 3 through 11 probe memory

Require: **curr**: the first (leftmost) index object,

key: the key to locate

Define: **curr**→**next**: a soft pointer,

curr→**next**[**i**]: the *i*-th probe of the soft pointer.

```
1: while curr→key != key do
2:   flag←0;
3:   for each curr→next[i] do
4:     if curr→next[i]→key < curr→key then
5:       continue; {rule out backward probes}
6:     end if
7:     if curr→next[i]→key ≤ key then
8:       curr ← curr→next[i]; {take a forward move}
9:       flag←1; break;
10:    end if
11:  end for
12:  if flag=0 then
13:    return NOT_FOUND; {all the probes are not useful}
14:  end if
15: end while
16: return FOUND;
```

locations to de-reference the soft pointer in the current key. Step 5 rules out backward probes, and Step 8 takes a forward skip. These steps ignore probes skipping beyond the searched key. Step 13 reports that the searched key not found if all the probes are not useful. Step 16 reports that the search is successful if the current key matches the searched key.

0.3.3 Key Insertion and Deletion

Soft lists and linearly ordered lists are structurally similar. When inserting a key to a soft list, the new key is inserted immediately after the key that is just smaller than the new key. This smaller key is referred to as the *immediate predecessor* of the new key. Deleting a key also involves the key's immediate predecessor. Therefore, the first step in handling an insertion or deletion is to find a key's immediate predecessor in a soft list.

The procedure of finding the immediate predecessor of a specified key is based on a search algorithm. Starting from the leftmost key, this procedure locates the *first* index

object whose soft pointer refers to no keys between the current key and the specified key.¹¹ For example, before inserting a new key 16 to the soft list in Fig. 4(b), the key immediately smaller than 16 must be found. Starting from key 10, the current key moves forward to key 15 via the first probe. At key 15, the two probes give keys 40 and 70, both of them are no smaller than key 16. As a result, this procedure reports key 15 as the immediate predecessor of 16. Algorithm 1 can be slightly revised for this purpose: Instead of reporting “NOT_FOUND” at Step 13, “curr” is reported as the immediate predecessor.

Inserting a new key or deleting a key changes the immediate predecessor’s soft pointer. Updating pointers out of place introduces pointer-update propagation, as described in Section 0.2.1. To deal with this issue, every index object in a soft list reserves a number of spare slots as *spare pointers*. When revising a pointer, changes are logged into these empty slots. Retrieving a soft pointer scans the spare pointers to find its most recent version. Incremental pointer logging is feasible because NOR flash is byte-addressable. This way, a soft pointer can be revised many times.

Whenever an index object runs out of spare slots, the turnstile encompassing this object can be rotated to refresh its spare slots. However, rotating a turnstile just to refresh one object’s spare slots may be premature garbage collection, because there may still be a lot of free space in the turnstile for new objects. Instead, we choose to rewrite the object out of place. Even though this could introduce pointer-update propagation, the propagation spreads slowly due to the spare pointers. For example, if every object has m spare pointers, then an object must be updated m^n times to propagate this update n objects away. Whenever necessary, a soft list can rotate turnstiles to completely stop this propagation.

Spare pointers and soft pointers have different design goals. However, they are not exclusive, and soft lists use them both. Using soft pointers avoids garbage-collection deadlock, while using spare pointers alleviates the propagation of pointer updates.

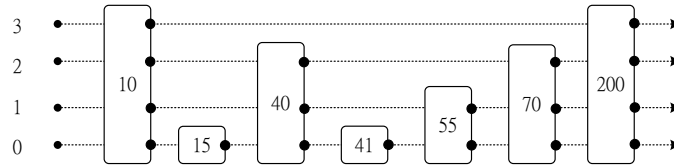


Figure 5: A multilevel soft list comprises four simple soft lists. Index objects hook on soft lists by means of soft pointers. Note that references of soft pointers to buddy objects are not drawn.

0.4 Multilevel Soft Lists: Scalability Enhancement

Multilevel soft lists organize multiple soft lists in parallel to provide better scalability. This section presents the design of multilevel soft lists, and then discusses strategies related to wear leveling, space allocation, and boot-up initialization. The last part of this section analyzes the performance of multilevel soft lists.

0.4.1 Structure of Multilevel Soft Lists

Because soft lists are structurally similar to linearly ordered lists, searching for a key in a soft list can degrade into a linear search in a worst case scenario. This study proposes combining multiple parallel simple soft lists into a multilevel soft list. Let a multilevel soft list have n parallel simple soft lists, where level 0 is the lowest level. An index object in this multilevel soft list has n soft pointers, one for each level. Whether or not an index object can hook on a certain level is controlled by a probability parameter p , where $0 < p < 1$. Let q_1, q_2, \dots , and q_{i-1} be randomly generated numbers whose values are between 0 and 1, and let $q_0=0$. An index object hooks on the level- i simple soft list only if $(q_0 \leq p) \wedge (q_1 \leq p) \wedge (q_2 \leq p) \wedge \dots \wedge (q_i \leq p)$ is true. In other words, the probability that an object hooks on the level- i soft list is p^i . Thus, the expected total number of objects hooking on the level- i simple soft list is $N \times p^i$, where N is the total number of keys in a multilevel soft list.

Figure 5 depicts a four-level soft list extended from the simple soft list in Fig. 4. This multilevel soft list is structurally similar to a skip list [22]. Index objects hook on soft lists

by means of soft pointers. Note that Fig. 5 illustrates only the references to target objects,¹³ and omits references to buddy objects. Let the *current level* be the level that is currently visited in a search session. Searching for a key in a multilevel soft list begins with the first (leftmost) key at the highest-level soft list, using the same search algorithm as that used in simple soft lists. If the searched key cannot be found in the current level, then the current level descends to the next lower level and continues the search from the current key.

Consider searching key 55 in the multilevel soft list depicted in Fig. 5. For brevity, the following discussion ignores any probe referring to a buddy object. This search begins with key 10 at the level-3 list. At level 3, key 200 is immediately next to key 10. Because key 200 is larger than key 55, the current key falls back to key 10 and the current level descends to the level-2 list. At level 2, the current key moves to key 40. The key immediately next to 40 is 70, which is again larger than key 55. Therefore, the current level descends to the level-1 list. At level 1, the searched key 55 is found next to the current key 40, and the search is reported successful.

The search algorithm for multilevel soft lists is based on Algorithm 1. One extra variable is required to indicate the current level, and searching for a key always starts from the first key at the highest level. Step 13 of Algorithm 1 is revised as follows: If the current level is not the bottom level, then move the current level downward to the next lower level. Otherwise, report NOT_FOUND. This search algorithm for multilevel soft lists can be revised to support insertion and deletion, analogous to revising the search algorithm for simple soft lists.

Multilevel soft lists inherit the advantages of both skip lists and soft pointers. High-level lists enable long-distance skips, while rightward moves at any level benefit from random forward skips. Another advantage to this approach is that multilevel soft lists require no extra expensive flash write operations for self-balancing, as they are a kind of randomized index structure.

0.4.2 Space Allocation and Wear Leveling

Buddy objects enable random forward skips, making it possible to speed up a search. Ideally, all the buddy objects of a soft pointer should be randomly distributed among all the keys in a soft list to create the largest possible separation between buddy objects. This coincides with the consideration of wear leveling in flash blocks. As each flash block individually endures a limited erasure cycles, typically 100 K cycles [15], uniformly erasing all flash blocks postpones the first appearance of worn-out blocks, keeping all blocks alive as long as possible. Thus, random placement of index object not only benefits search performance, but also extends flash lifetime.

This study proposes a two-level space allocation policy. When a new object is to be written to flash, a block-level allocation policy randomly selects a block to accommodate the object. If the selected block has no free space, it becomes a victim block for erasure. As described in Section 0.3.1, to evenly erase all the blocks in a turnstile, the garbage collection procedure rotates the turnstile until the victim block is erased. However, if the victim block is far from the spare block, then garbage collection must rotate the turnstile several times, unnecessarily erasing many non-victim blocks. Since the block-level policy is random selection, this already gives all the blocks in a turnstile an equal chance of being erased. So, instead of rotating turnstiles, garbage collection first shifts all the valid objects from the victim block to the spare block, and then erases the victim block into a spare block.

At high levels in a multilevel soft list, each forward move is expected to skip over a large number of keys. However, unconditionally taking random forward skips could prematurely decrease the current level. Consider searching key 70 in Fig. 5. Let the current key be 10 and the current level be 2. Suppose that the soft pointer at this position refers to a target object 40 and a buddy object 41 (this reference is not drawn in Fig. 5). Even though key 41 is ahead of key 40, moving the current key forward to key 41 can demote the current level to level 1, prematurely decreasing the future skip distance. Therefore, the probe to key 41 is useless.

This study proposes an object-level space allocation policy to reduce the possibility of

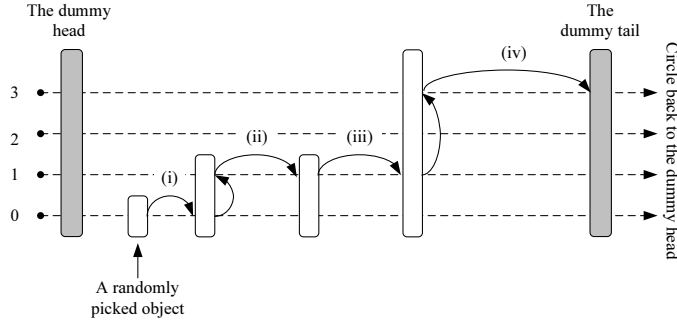


Figure 6: A scenario of initializing a multilevel soft list.

producing these useless probes mentioned above. Let B be the flash block size. When writing a new level- x object to flash memory, the block-level allocation policy first chooses a flash block. The object-level allocation policy then searches the free space starting from the block offset $\lceil B(1 - p^x) \rceil$ toward the end of this block. This search circles back to the beginning of the block, and aborts if the initial block offset is encountered a second time. The new object is written to the first encountered free space. This object-level allocation policy tries to cluster objects on the same level in nearby block offsets, reducing the chance of probing lower-level buddy objects.

0.4.3 Boot-Up Initialization

The soft pointers approach does not need to store an address mapping table in volatile memory (i.e., RAM), saving the time required to scan the flash memory to re-build the mapping table after powering on. Initializing a multilevel soft lists locates the first index object on each level. A brute-force approach is to fix all these first index objects at convenient locations in NOR flash. However, nailing down objects in flash memory can damage wear leveling in flash blocks.

A multilevel soft list consists of a dummy head and a dummy tail as the leftmost object and the rightmost object, respectively. These two objects hook on every level. Let the dummy head be immediately next to the dummy tail. The initialization process locates the dummy head's residence in NOR flash, as follows: First pick up an index object from a randomly selected memory location in NOR flash as the current key. Let the current

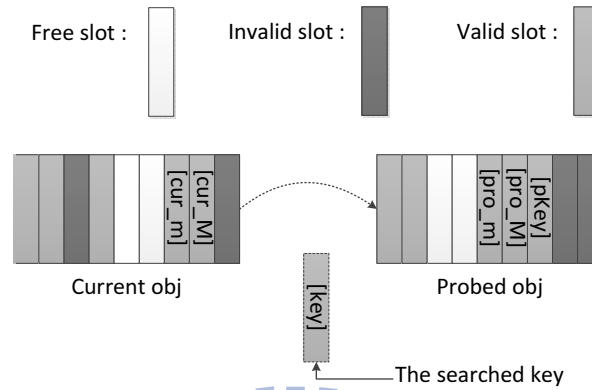


Figure 7: The state of de-referencing a soft pointer's probe

level be the highest level this object hooks on. Next, repeatedly move the current key rightward, and climb to high-level lists whenever possible. Upon reaching the dummy tail, take one further step forward to the dummy head. This procedure takes advantage of the long-distance skips provided by high-level lists, greatly speeding up the initialization time.

Figure 6 shows an example of initializing a multilevel soft list. This procedure first randomly picks up an object at level 0. Step (i) moves the current key to a level-0 object, and then increase the current level to 1. Step (ii) moves the current key to another level-1 object. Step (iii) moves the current key to a level-3 object and then promotes the current level to 3. Step (iv) brings the current key to the dummy tail. At this time, the dummy head is one object away. Note that in this process, any rightward moves via target objects or buddy objects can be taken except for those demoting the current level.

0.5 Fat List: Page-Oriented Soft Lists

Fat list let many keys contained in an object in order to reducing the frequencies of structural operations. This section presents the design of fat list, including the additional operation according to fat list.

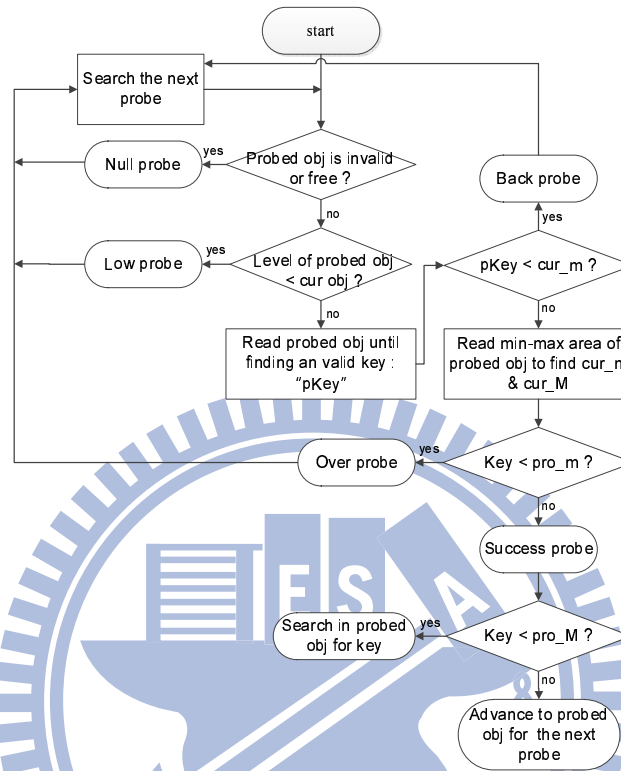


Figure 8: The flow chart of de-referencing a soft pointer's probe

0.5.1 Fat Objects

Because the object in soft lists has only one key, a structural modification on every update is necessary. Therefore, a fat object consists of m slots, bitmaps mapping to slots, and a pointer pool.

The slots in an object are free space for keys. The maximum and minimum keys would put together in min-max area in slots in order to get the object's range with less read. If the referenced key is in the object range, the referenced key would put in a free slot. Then if there is already the same key in other slots, mark it into invalid by bitmaps. The motivation to have many free slots in an object is to let keys have free space to update. If the different keys number in an object is too many, there would no free space for every key. Therefore, we restrict that every object could have only n different keys, and n is less than m . In the rest of this paper, we use the term key number to refer to the number of different keys in an object, and n is the restriction of different key number. We also use

the term slots number to refer to the free space for keys in an object, and m is the total number of slots.

Although the min-max area can get the object's range with less read, the updating of maximum and minimum keys makes the min-max area larger and increases the time of reading min-max area. In fact, the range of every object does not overlap. Reading whole min-max area is not needed when de-referencing a soft pointer and analyzing the object whether can skip forward or not. Figure 7 shows the state while de-referencing a soft pointer's probe, and the flow chart of de-referencing a soft pointer's probe is shown in Figure 8. If the probed object is free or invalid, the data in this object are useless and reading in this object is not needed. If the level of the probed object is lower than the current object, skipping to this object is not allowed. This kind of object even need not to read the keys in slots. Then, if the probed object is not free, invalid, and lower level, reading keys in this object is needed to judge whether to skip forward or not. Reading until finding the first valid key, $pKey$, in min-max area is enough to determine that this object could not skip forward if $pKey$ is smaller than the minimum key in current object. However, if $pKey$ is larger than the minimum key in current object, reading whole min-max area is needed in order to get the minimum key and analyze the range of the probed object is larger than the searched key or not to determine whether could skip forward or not. Using the strategy to analyze the objects the soft pointer reference to can decrease the time of searching the target key.

In multilevel soft lists, free pointer space for every level pointer is fixed. Whenever update a pointer with no free pointer space in that level but free pointer space in other levels, the object with many unwritten free pointer space would update in other free space. The objects in fat list have pointer pool, let every level pointer all put together. Therefore, the object would be updated when run out of free pointer space. Although the read caused by pointer would increase, the frequencies of structural operations would reduce, and the write and erase would decrease. Because a read spends time far less than write and even read, the tradeoff is worth.

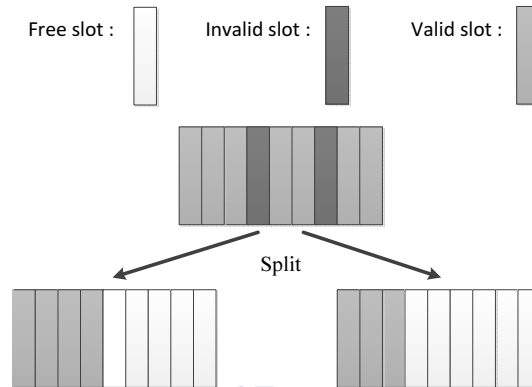


Figure 9: An operation on delayed split with $m = 9$, $n = 6$.

0.5.2 Delayed Split of Fat Objects

By the restriction of different key number, the operation of splitting the object will be performed when inserting a new key and then causing the key number exceeding the restriction n . As Figure 9 shown, the operation to split the object is to copy half of the number of the key to each of another two new object.

However, if the split operation is performed whenever key number exceed n , it would not only cause frequent structural modifications but also waste slots by copying the keys. The way to prevent the above circumstances is to delay the split operation until the slots in an object are full of keys and key number exceed n . When the slots are full, the keys in the slots should be copied to another object originally. Therefore, the split operation is performed when slots full would not cause extra writes of slots and also reduce the structural modifications.

The keys in the slots should also be copied to another object while reclaiming free space, but the split operation would not be performed even if the key number in the object exceed n . This is because if the chosen victim has many objects with key number exceeding n , it may cause garbage-collection deadlock. Moreover, the data in the object is cold so that the slots have not been full until being reclaimed. The cold data should not occupy too many slots with rare using. According to above reason, the split operation would not be performed while reclaiming free space.

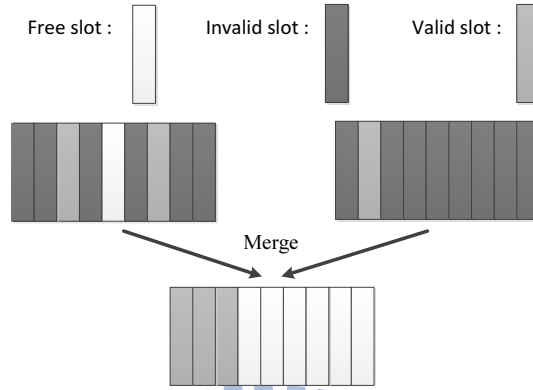


Figure 10: An operation on delayed merge with $m = 9, n = 6$.

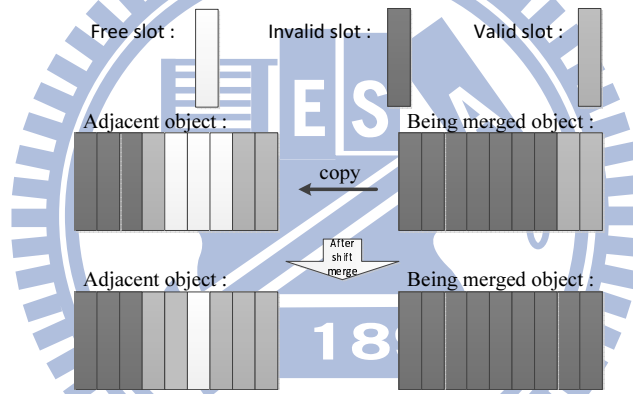


Figure 11: An operation on shift merge with $m = 9, n = 6$.

0.5.3 Lazy Merge of Fat Objects

In order to reduce flash space utilization and thus improve free-space reclaiming efficiency, the space utilization in objects should be improved. Therefore, the operation of merging two adjacent object will be performed when both two adjacent objects have poor space utilization, and the definition of poor space utilization is the key number in an object is less than half of n . As shown in Figure 10, the operation to merge two adjacent objects is to copy all the data in the two objects into another new object.

Just as the problem in splitting object, if the merge operation is performed whenever key number is less than half of n , it would cause frequent structural modifications. The way to prevent this is to delay the merge operation until not only the slots in an object are full of keys but also reclaiming free space, and key number is less than half of n . While reclaiming

free-space, the data in the poor space utilization object is cold so that the object have not²¹ been dealt with. In order to prevent cold data occupying many slots, the merge operation is performed while free-space reclaiming. While the slots in an object are full, the data in the object may not be cold data. However, performing merge operation on two poor space utilization objects is necessary in order to reduce flash space utilization. Additionally, objects should at least use up all their empty slots before being merged. Therefore, as long as objects have poor space utilization, the merge operation should be performed without wasting slots. Note that if objects contain cold data and cannot use up all their empty slots upon garbage collection, then these objects will be merged anyway.

However, if the adjacent object which is chosen to be merged still has many free slots that have not been used, this situation will cause wasting slots. Therefore, as Figure 11 shown, if the adjacent object has enough free slots number for the key in the being merged object, the key and data in the being merged object are copied to the adjacent object. This operation is defined as shift merge. Shift merge could reduce flash space utilization by not to merge to another new object.

The other way to prevent objects having poor space utilization is to borrow key from the adjacent objects without poor space utilization. This may increase the opportunity to improve the space utilization in objects. Nevertheless, borrowing keys waste slots. This may cause extra writes and decrease free-space reclaiming efficiency. Therefore, borrowing keys would not be performed.

0.5.4 Processing Range Queries

When dealing with range queries, multilevel soft list will first find the first data location, and then read the data along the lowest level until finding the last data. The data except the first data need not be found by searching from head. However, search every key and data follows a pointer. That would cause many read times and be slow when dealing with range queries.

Fat list could find much more keys in an object and follows a pointer, but there is some

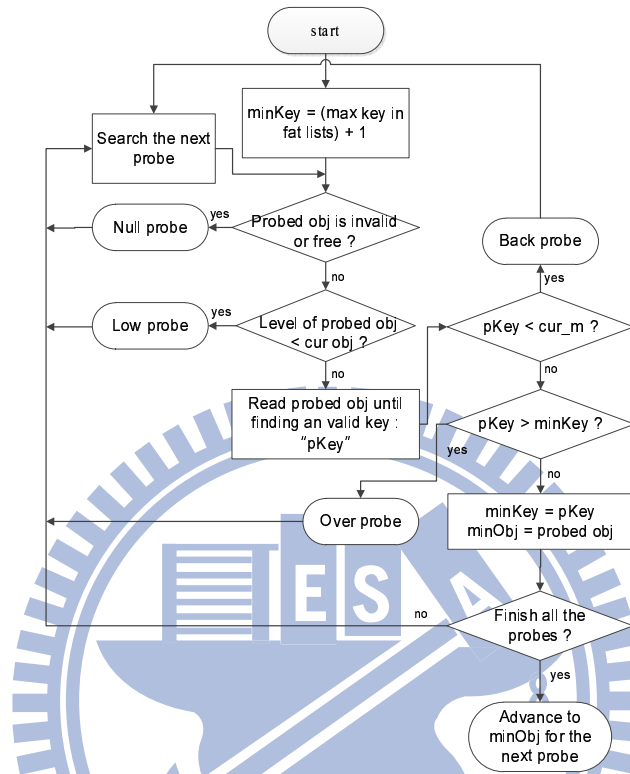


Figure 12: The flow chart of de-referencing a soft pointer's probe in range query

extra reads of invalid data while reading min-max area. Except finding the first key of the range, reading every soft pointer of an object should also read all the objects the soft pointer reference to in order to confirm which object is exactly the next object while finding the keys in the range. This overhead is in soft lists, too. However, different from the search mechanism in fat lists, reading whole min-max area is not necessary when the keys in the probed object are larger than that in the current object. As shown in Figure 12, the flow after confirming the probed object is not free, invalid, and lower level is different from the original search flow. Because of the characteristic of not overlapping range in fat lists, reading the first valid key in min-max area in these objects is sufficient to weed the smaller-key objects out and find the object with minimum key in the larger-key objects to confirm which is the next object. Therefore, the advantage of fast judgement could cover the disadvantage of reading extra invalid data. Because fat lists could find much more keys in an object and just read a pointer, fat list is faster than multilevel soft list when dealing with range queries.

Geometry		Timing	
Word size	2 bytes	Word read	110 ns
Capacity	1M words	Word write	80 us
Block size	32K words	Block erase	0.6 s
Block endurance	100K cycles		

0.6 Experimental Results

0.6.1 Experimental Setup and Performance Metrics

We implemented a simulator to evaluate the performance of fat lists. This simulator adopts the specification of a real-life NOR flash [15] shown in Table 0.6.1. Unless explicitly specified, the experiments in the rest of this paper use the following default settings for fat lists: the turnstile size is 8, the maximum level is five, the slots number is 40, the key number is 20, the pointer pool size is 7, and the object size is 353 byte according to the above settings. The probability parameter p is 0.25, as suggested in [22].

Our experiments consisted of two parts: a micro-benchmark and a macro-benchmarks. The micro-benchmark is divided into four phases: insertion, update, query, and deletion. This micro-benchmark investigates how fat lists perform under different types of monotonous access patterns. On the other hand, the macro-benchmarks are based on workloads gathered from real-life applications, showing diverse access patterns. This study compares fat lists against a tree-based index structure in NOR flash, which the following section describes in greater detail.

Our experiments evaluated the performance of index structures in terms of the total time of word reads, word writes, and block erasure calculated based on the read, write, and erase times and timing characteristics in Table 0.6.1. The time contributed by pure operations and free-space reclaiming are counted separately to tell them apart.

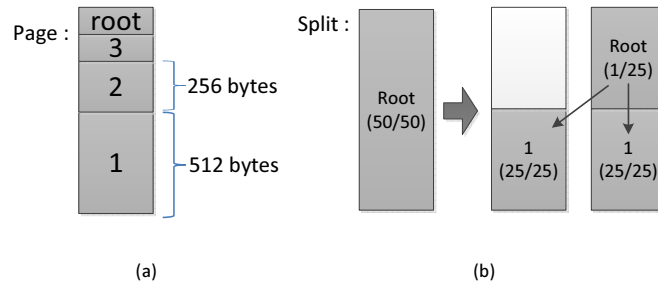


Figure 13: Layout of μ -tree with page size 1024 bytes. (a) The size distribution in different level node with 4 level in a page. (b) The split operation and increase height happened when the root node is full.

0.6.2 μ -Trees in NOR flash

μ -tree is similar to B+-tree. To minimize the write times which are caused by writing all the nodes from root to leaf while updating a leaf node, the path from root to leaf is written into a page. Since the upper level the node is in, the fewer data the node has, the size of node is depend on the level of the node and the height of whole tree. As shown in figure 13(a), the size of the node is half of the size of the next lower level node except root node. If the tree height is one, the whole page is a node. In figure 13(b), supposing that the only node of the tree is full, the node will split into two node and create a new root, and the size of nodes is shrank to half. Once the root is full again, the root will split and the size of new root and next lower level node is also shrank to half of the size of the original root, and so on.

There is no balancing operation when deleting keys. In our simulation of μ -tree, in order to maintain the structure as B+-tree, if no key left in the node after deleting a key, borrowing a key from adjacent node is necessary.

In the simulation of μ -tree, there is no spare pointer. If there are spare pointers in every node, the nodes along the path will not all rewrite to a new page while updating a leaf node. Therefore, the leaf node and the upper level nodes in its path may not be written in the same block. That may cause garbage-collection deadlock because of writing more node than expected. To prevent this, there is no spare pointer in μ -tree in order to rewrite the whole path to a page.

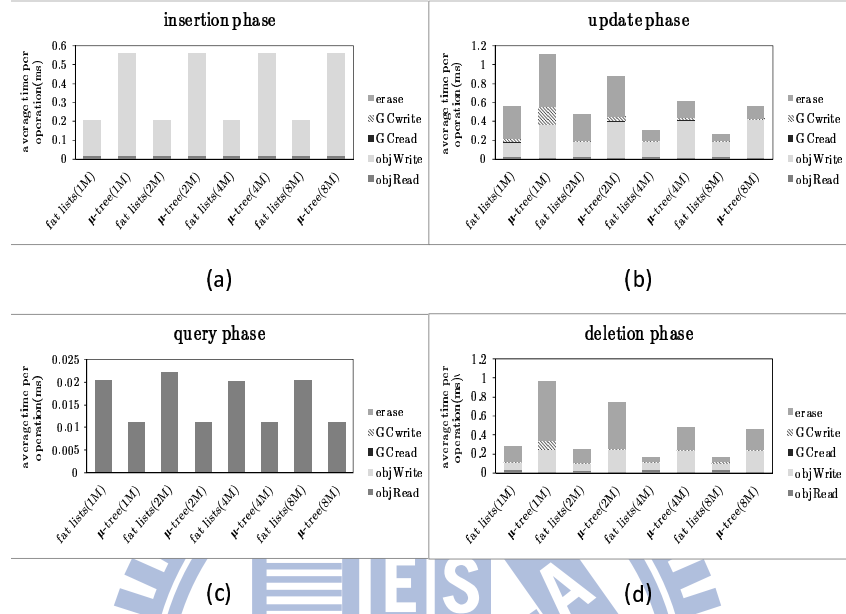


Figure 14: The micro-benchmark results of fat lists and μ -tree in different flash size 1 mega-byte, 2 mega-byte, 4 mega-byte, and 8 mega-byte (a) insertion phase (b) update phase (c) query phase (d) deletion phase

0.6.3 Micro-Benchmark Results

Test Procedure

The micro-benchmark test procedure in this study consists of four phases: insertion, update, query, and deletion. The NOR flash is entirely empty before this benchmark. The first phase sequentially inserts 25,000 consecutive keys. The second phase performs 800,000 key updates using a Gaussian random variable for key selection, forming a temporal locality in the access pattern. The third phase performs 800,000 key queries using the same random variable for key selection. The final phase randomly removes all the keys. The mean and the variance of the Gaussian random variable are the median of all the keys and one-sixth the total number of keys, respectively.

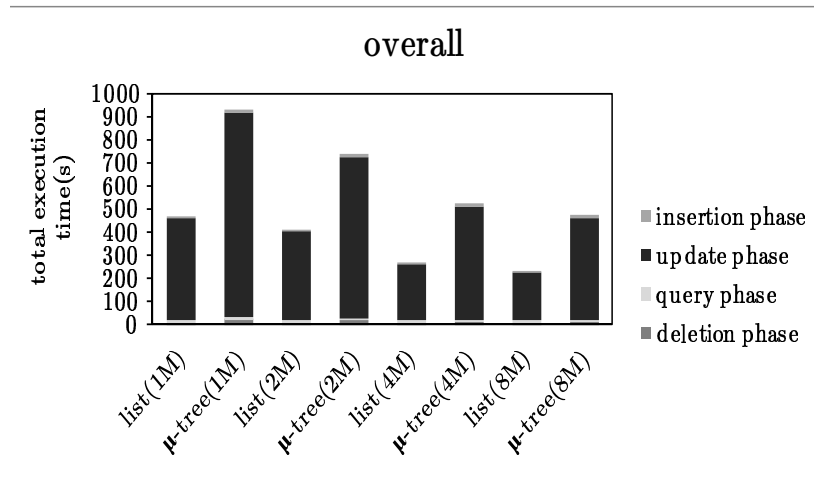


Figure 15: The total execution time of the micro-benchmark in different flash size 1 mega-byte, 2 mega-byte, 4 mega-byte, and 8 mega-byte

Fat Lists versus μ Trees

Figure 14 shows the micro-benchmark results in different flash size. Before discussing these results, recall that NOR flash is very slow on write and erase but extremely fast on read. The page size of μ -tree is 512 bytes in order to contain all data after the insertion phase. In insertion phase, the flash size does not affect the performance on fat lists and μ -tree, because the minimum need of flash size to build up the list and tree is less than 1M. Therefore, fat lists and μ -tree does not trigger any garbage-collection activities. The read time of μ -tree is less than fat lists because the query performance in μ -tree is determined by its height. μ -tree is also a balanced tree, so the increase of height is very slow. For this reason, its hard to outperform μ -tree in query. However, the write time of μ -tree is much more than fat lists because μ -tree has to rewrite all path on every structural modification of leaf node. Since NOR flash is much more slower on write than on read, fat lists outperforms μ -tree in insertion phase.

Although there is no garbage-collection activity in insertion phase, the minimum need of space to build up μ -tree is more than that of fat lists after inserting all keys. This is because the data in μ -tree is all contained in leaf node, and every leaf node in μ -tree occupy a page. A leaf node with size 256 byte in μ -tree need 512 bytes, and an object size in fat lists is 353 bytes including 20 data, 20 keys, a 20-bit bitmap, and a pointer pool of size 7,

two words each. The leaf nodes in μ -tree contain a little bit more data than objects in fat lists, but the occupied size is much larger. Therefore, to build up μ -tree needs more space than fat lists. In the same flash size, the free space of μ -tree would less than fat lists.

In update phase, Both fat lists and μ -tree experience garbage-collection activities. As the flash size decrease, the performance of fat lists and μ -tree also decrease. The read time of μ -tree is still less than fat lists, but the write and erase time of μ -tree is much more than fat lists. This is not only because μ -tree has to rewrite all path on every structural modification but also because the free space of μ -tree is less than fat lists. Rewriting all path increase write time, and less free space increase erase time. Therefore, fat lists outperforms μ -tree in update phase.

μ -tree outperforms fat list in query phase. Query in balanced tree is fast, and its hard to outperform tree in query. This is a tradeoff between the performance of read-only queries and read-write operations. Because read operation is much faster than write and erase, we choose to sacrifice the performance of read-only queries and improve the performance of read-write operations.

The deletion phase is also a read-write phase. When deleting keys, the structural modifications are less than update. However, fat lists still outperforms μ -tree. Figure 15 shows the total execution time of the micro-benchmark in different flash size. Although μ -tree outperforms fat list in query phase, fat lists is still two times faster than μ -tree in the total execution time because NOR flash is fast on read. Overall, this micro-benchmark shows that fat lists is much faster than μ -tree.

Maximum Level and Turnstile Size

This section analyzes fat lists performance under different settings. The first experiment evaluates fat lists using different maximum levels. Figure 16 presents the experimental results of the micro-benchmark. In insertion phase, the difference of write time between different settings of maximum level is small, but the read time decrease as the maximum level increase. This is because the higher level can skip more distance than lower level. Therefore, the difference of total time between different settings of the maximum level

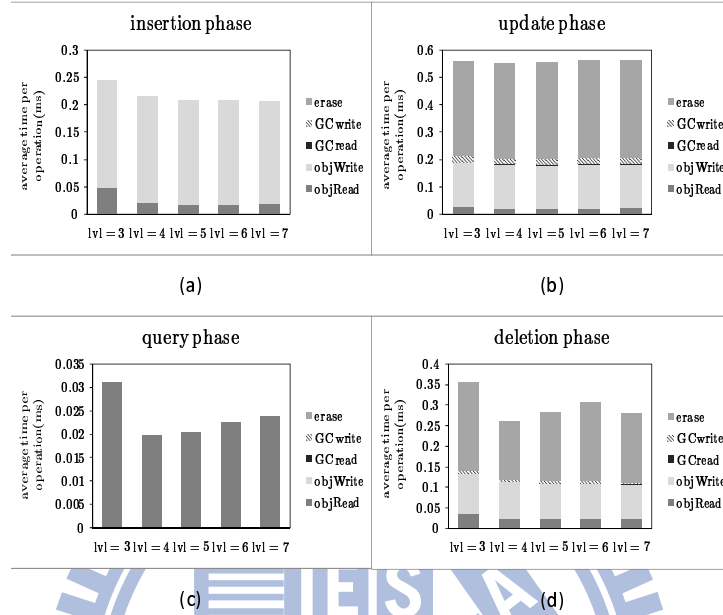


Figure 16: The micro-benchmark results of fat lists with different settings of maximum level (a) insertion phase (b) update phase (c) query phase (d) deletion phase

determined by the read time in insertion phase. In update phase, the performance is better with maximum level setting to 4 and 5. The performance with maximum level setting to 3 is worse because the maximum level is too small that it spends much more time in query data location. As the maximum level increasing, the size of pointer pool increases, and object size increases, too. Therefore, an object is larger, the consumption of space is faster. The frequency of garbage-collection activities also increases. Thus, the setting of larger maximum level performs worse because of increasing garbage-collection activities frequency. In query phase, the performance is better with maximum level setting to 4, and the result is different with that in insertion phase. Because there is more keys contain in an object in fat lists, the object number is not many. As we use the mechanism of [22] to allocate the number of objects in different level, the high level object amount is few and even none except head and tail objects. The benefit of high level to skip farther is useless, and even worse this increases the read time searching from highest level to lower level. After garbage-collection frequency, the read time increases to find the target, and this also reveals the drawbacks of few high level objects. In deletion phase, the tendency of read time is as in query phase, but the write time decrease as the maximum level increase. As mentioned in update phase, the increasing of maximum level increases the size of pointer pool and object

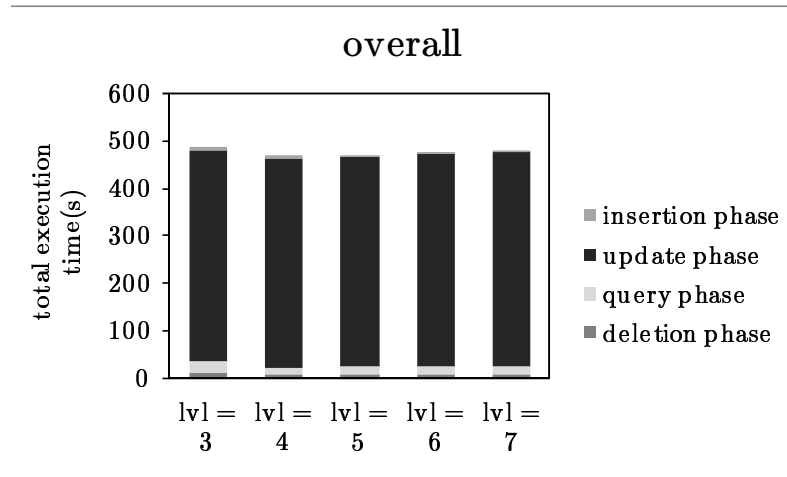


Figure 17: The total execution time of the micro-benchmark with different settings of maximum level

size. When the maximum level is small, the size of pointer pool is small. Therefore, the spare for updating pointers is less, and that will cause frequent object rewrites because of no spare to update pointer. That is, in small maximum level, small object size decrease the frequency of garbage-collection activities, but small size of pointer pool increase frequency of rewriting objects and increase write time. There is less operation in deletion phase, so the increasing frequency of rewriting objects because of small size of pointer pool reveals. Figure 17 shows the total execution time of the micro-benchmark with different settings of maximum level. The better choice of setting maximum levels are 4 and 5 in this experiment.

Figure 18 shows the results of evaluating fat lists using different turnstile sizes. As the turnstile size increases, the read time increases in four phases, and especially obvious after update phase. This is because the probability of probing an invalid object increases as the space utilization in NOR flash decreases. Thus, de-referencing a soft pointer may require extra probes to find a valid object and successfully skip. However, the the setting of small turnstile size may not perform the benefit of using soft pointers that can randomly skip farther. This phenomenon is shown in the query phase. Even so, the better performance in update phase is turnstile size being set to 8. Because every turnstile has one spare block, larger turnstiles will result in fewer spare blocks and low space utilization in NOR flash. Therefore, the write and erase time decreases as the turnstile size increases. Figure 19 shows the total execution time of the micro-benchmark with different settings of turnstile

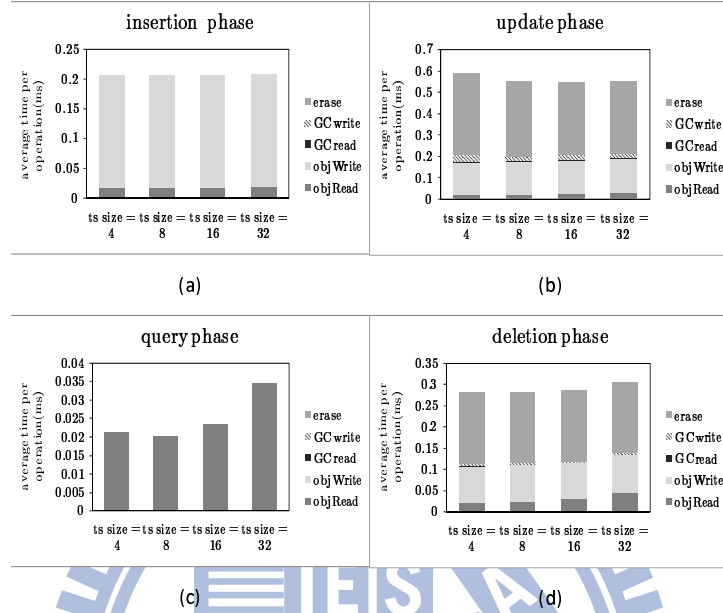


Figure 18: The micro-benchmark results of fat lists with different settings of turnstile size (a) insertion phase (b) update phase (c) query phase (d) deletion phase

size. The better choice of setting turnstile sizes are 8 and 16 in this experiment. Based on the above results, the recommended maximum level and turnstile size are 5 and 8, respectively.

Overhead of De-referencing Soft Pointers

De-referencing a soft pointer probes objects in a turnstile, but not all probes produce useful results. This section investigates the overhead caused by these extra probes.

To assist our discussion, we first define different types of probing results: a null probe points to an invalid object or free space in NOR flash, a back probe goes to an object whose key is smaller than the current key, a low probe points to an object that does not hook on the current level, and an over probe refers to an object whose key is larger than the searched key. None of these probes are useful. All the other probes are useful, and called successful probes. Figure 20 illustrates the different types of probes.

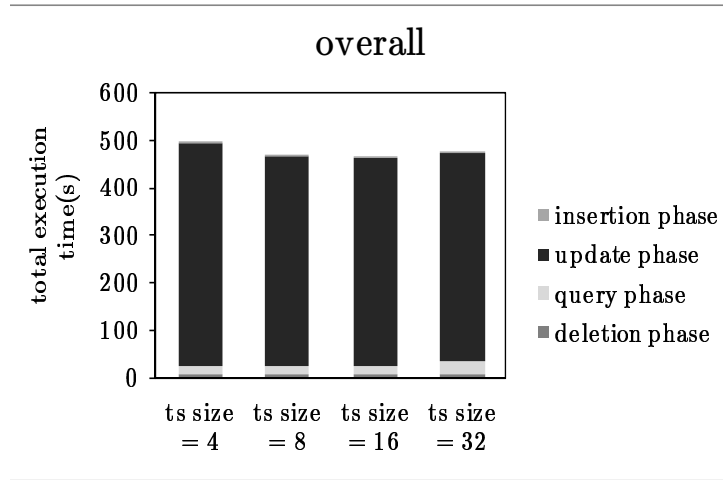


Figure 19: The total execution time of the micro-benchmark with different settings of turnstile size

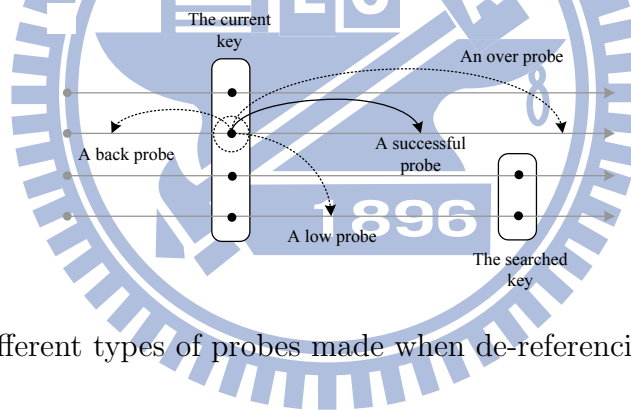


Figure 20: Different types of probes made when de-referencing a soft pointer.

This part of our experiment is to analyze the soft pointers in phase 3 in the micro-benchmark. Table 0.6.3 shows these results. The rows titled “pointers visited” and “total probes,” show how many soft pointers are de-referenced and how many probes are made during the test, respectively. The rest of the rows show the percentage of probes separately to differentiate between probe types.

Almost every soft pointer has a successful probe except the soft pointer at the last of every level before finding the searched key. The maximum level is higher, the soft pointers with no successful probe is more. In maximum level setting to 1, percentage of successful probes is 1 because there is just one level and must find an object to skip to before finding searched key. Therefore, as the maximum level decreases, the percentage of successful probes increases.

	Max. levels			
	5	4	3	1
Pointers visited	13082578	13130214	22495212	88385632
Total probes	34529843	28286792	35443585	106887758
successful probes	0.775	0.837	0.940	1
low probes	0.020	0.017	0.001	0
over probes	0.676	0.487	0.192	0.057
back probes	0.364	0.233	0.108	0.043
null probes	0.804	0.580	0.334	0.109

Now consider the unsuccessful probes. As mentioned in Section 0.4.2, fat lists adopt an object-level space allocation policy, writing objects at the same level to nearby offsets of the blocks in a turnstile. The percentage of low probes are small, proving this policy successful. Null probes, back probes, and over probes are the most common types of unsuccessful probes. Provided that valid objects are randomly distributed in the entire NOR flash, the total number of back probes and over probes will be proportional to the total number of pointers visited. On the other hand, the total number of null probes is subject to not only the total number of pointers visited but also the space utilization in NOR flash. The lower the space utilization is, the more null probes there will be.

The discussion above shows that the total number of extra probes is a function of the total number of successful probes and the space utilization in NOR flash, meaning that overhead is manageable.

Initializing Speed

This experiment evaluates how many word reads the initialization procedure requires to initialize a fat lists. The initialization procedure is inserted between the update phase and the query phase of the micro-benchmark because this is when the fat lists contains the largest number of keys. This test was conducted under different maximum levels and turnstile sizes, while the total number of keys remained at 25,000.

The results in Table 0.6.3 show that the initialization overhead drastically decreased as the maximum level increased. This is because the initialization procedure escalates

TS sizes	Max. levels			
	1	3	4	5
8	150.59	37.95	30.47	15.07
16	184.03	58.74	30.47	20.13
32	266.75	66.44	33.22	25.30

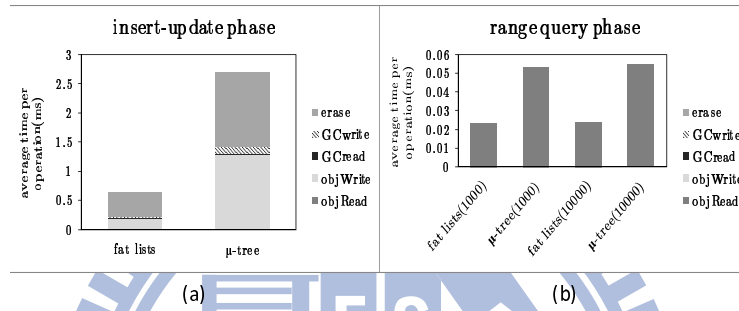


Figure 21: The macro-benchmark results of fat lists and μ -tree with different times of range query (a) insertion phase (b) range query phase

the current level for long-distance skips. Conversely, the initialization overhead increases when the turnstiles are large. This is because space utilization in NOR flash is inversely proportional to the turnstile size. Decreasing the space utilization increases the probability of making null probes when de-referencing soft pointers.

These results indicate that the maximum level has a much greater influence on the initialization overhead than the turnstile size. When the turnstile size and the maximum level are 8 and 5, respectively, initializing a fat lists of 25,000 keys takes only 15.07 microseconds.

0.6.4 Macro-benchmark Results

This part of our experiment considers a real-life workloads. This workload is collected from 54 sensors deployed in Intel Berkeley Research lab between February 28th and April 5th, 2004. The Mica2Dot sensors with weather boards collected the weather data once every 31 seconds and got a log of about 2.3 million readings. We filtered out the data of temperature and modified it to be appropriate for index operation.

This experiment use the following settings for fat lists: the turnstile size is 8, the maximum level is five, the slots number is 40, the key number is 30, and the pointer pool size is 7. The probability parameter p is 0.25, as suggested in [22]. The page size of μ -tree is 512 bytes.

First, we insert the temperature data according to the order of collecting data. If the same value data was inserted again, we regarded it as update. After inserting all the data, we randomly produced the start keys and ranges from the temperature data to range query.

Figure 21 shows the results of fat lists and μ -tree. In insertion phase, the read time of fat lists is more than μ -tree since the query performance in μ -tree is depend on tree height and the tree height of μ -tree grows slow. However, the write and erase time of μ -tree is much more than fat lists because every leaf node in μ -tree occupies a page which is much larger than an object in fat lists. Therefore, the flash space utilization of μ -tree is more than fat lists, and garbage-collection in μ -tree is much more frequent than fat lists. On the other hand, μ -tree need to rewrite all path on every node update, so the write time of μ -tree is more than fat lists.

In range query phase, the performance of μ -tree is worse than fat lists no matter in 1000 times or 10000 times query. This is because μ -tree need to read from root on reading every range query node after finding the start key, and fat lists just need to read along the lowest level after finding the start key. As long as the overhead of reading the objects the soft pointer reference to in fat lists is less than reading the whole path in μ -tree, the performance of fat lists would be better than μ -tree in range query. Therefore, the read time of fat lists is still much less than μ -tree.

0.6.5 Discussion

The evaluation results in prior sections present the performance characteristics of fat lists. Using a large maximum level could increase the distances of skips, but if the high level object amount is so few that it could not skip long distance and also increases the search time from high level. Avoiding using very large turnstiles reduces the total number of useless probes,


but using very small turnstiles would result in too many spare blocks that would seriously³⁵ increase the garbage-collection frequency. Therefore, the setting of maximum level and turnstile size is important and depend on the workload.

0.7 Conclusions

Dealing with crucial limitations on computational resources is a fundamental design issue in embedded devices. Efficient data indexing not only provides fast data retrieval, but also prolongs battery life. Due to the write-once nature of flash memory, a major challenge of data indexing in flash memory is that data updates and pointer updates recursively trigger further updates. Previous studies tackle this issue using logical pointers, at the cost of large RAM-space requirements and a lengthy initialization scan. This study introduces a new pointer design, called soft pointers, and a novel index structure, called fat lists, that uses these soft pointers. A soft pointer allows de-referencing to probe a bounded number of physical locations in NOR flash. As a result, data objects can be moved around in NOR flash without invalidating a pointer, largely simplifying space management in NOR flash. Even better, the probes made by de-referencing a soft pointer provide opportunities for forward random skips in soft lists, greatly speeding up search operations. By enlarging the index objects, the frequency of structural modifications reduces and the speed of range query increases. The strategies of delayed split and lazy merge not only reduce structural modifications but also improve object space utilization and thus improve garbage-collection efficiency.

This study examines the performance characteristics of fat lists using a series of experiments based on a synthesized workload and a real-life workload. Results show that fat lists, taking advantage of very fast NOR flash reading but extremely slow writing and erasing, achieve a good performance for read-write operations. More importantly, fat lists save precious erasure cycles of flash blocks and extend the lifespan of flash memory.

Bibliography

- 
- [1] A. Hunter, “A Brief Introduction to the Design of UBIFS,” http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf, 2008.
- [2] B. Pfaff, “Performance Analysis of BSTs in System Software,” ACM SIGMETRICS Performance Evaluation review, Vol. 32, Issue 1, 2004.
- [3] C. H. Wu, T. W. Kuo, and L. P. Chang, “An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems,” ACM Transactions on Embedded Computing Systems, Volume 6, Issue 3, 2007.
- [4] C. H. Wu, T. W. Kuo, and L. P. Chang “The Design of efficient initialization and crash recovery for log-based file systems over flash memory,” ACM Transaction on Storage, Volume 2, Issue 4, 2006.
- [5] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J. Kim, “A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications,” ACM Transactions on Embedded Computing Systems, Vol. 7, issue 4, 2008.
- [6] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh, “Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices,” In Proceedings of the 35th International Conference on Very Large Data Bases, 2009.
- [7] D. W. Kang, D. W. Jung, J. U. Kang, and J. S. Kim, “ μ -tree: an ordered index structure for NAND flash memory,” in Proceedings of the 7th ACM/IEEE International Conference on Embedded Software, 2007
- [8] E. Gal and S. Toledo, “A Transactional Flash File System for Microcontrollers,” in Proceedings of the USENIX Technical Conference, 2005.

- [9] F. Buchholz, "The Structure of the Reiser File System," <http://homes.cerias.purdue.edu/~florian/reiser/reiserfs.php>, 2006.
- [10] J. Katcher, "PostMark: A New Filesystem Benchmark," Technical Report TR3022, Network Appliance, <http://www.netapp.com/techlibrary/3022.html>, 1997.
- [11] K. S. Yim, J. H. Kim, and K. Koh, "A Fast Start-Up Technique for Flash Memory Based Computing Systems," in Proceedings of the ACM Symposium on Applied Computing, 2005.
- [12] L. P. Chang and T. W. Kuo, "Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation," ACM Transactions on Storage, Volume 1, Issue 4, 2005.
- [13] O. Rodeh, "B-trees, Shadowing, and Clones," ACM Transactions on Storage, Vol. 3, Issue 4, 2008.
- [14] S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. W. Park, and H. J. Song, "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation," ACM Transactions on Embedded Computing Systems, Vol 6, Issue 3, 2007.
- [15] Samsung Electronics Company, "K8C1215ETM 32M x16 MLC NOR Flash Data Sheet," 2006.
- [16] Samsung Electronics Company, "K9GAG08U0M 2G * 8 Bit MLC NAND Flash Memory Data Sheet," 2006.
- [17] S. Lin, D. Zeinalipour-Yazti, V. Kalogeraki, D. Gunopulos, W. A. Najjar, "Efficient Indexing Data Structures for Flash-Based Sensor Devices," ACM Transactions on Storage, Volume 2 , Issue 4, 2006.
- [18] S. Nath, and A. Kansal, "FlashDB: Dynamic Self-Tuning Database for NAND Flash," In Proceedings of the 6th international Conference on information Processing in Sensor Networks, 2007.
- [19] S. Lee, and B. Moon, "Design of Flash-Based DBMS: an In-Page Logging Approach," In Proceedings of the 2007 ACM SIGMOD international Conference on Management of Data, 2007.

- [20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, “Introduction to Algorithms,” The MIT Press, 1990.
- [21] Y. Li, B. He, Q. Luo, and K. Yi, “Tree Indexing on Flash Disks,” In Proceedings of the 2009 IEEE international Conference on Data Engineering, 2009.
- [22] W. Pugh, “Skip Lists: A Probabilistic Alternative to Balanced Trees,” Communications of the ACM, Vol. 33, No. 6, 1990.

