Journal of Systems Architecture 60 (2014) 357–371

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Memory efficient and scalable address mapping for flash storage devices

Young-Kyoon Suh [a,*], Bongki Moon [b], Alon Efrat [a], Jin-Soo Kim [c], Sang-Won Lee [c]

[a] Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA
[b] Department of Computer Science and Engineering, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul, Republic of Korea
[c] School of Information and Communication Engineering, Sungkyunkwan University, 2066 Seobu-Ro, Jangan-Gu, Suwon-Si, Gyeonggi-Do, Republic of Korea

A R T I C L E I N F O

Article history:
Received 14 July 2013
Received in revised form 4 December 2013
Accepted 13 January 2014
Available online 22 January 2014

Keywords:
Flash memory
I/O request
Extent mapping
Canonical extent

A B S T R A C T

Flash memory devices commonly rely upon traditional address mapping schemes such as page mapping, block mapping or a hybrid of the two. Page mapping is more flexible than block or hybrid mapping without being restricted by block boundaries. However, its mapping table tends to grow large quickly as the capacity of flash memory devices does. To overcome this limitation, we propose novel mapping schemes that are fundamentally different from the existing mapping strategies. We call these new schemes *Virtual Extent Trie (VET)* and *Extent Mapping Tree (EMT)*, as they manage mapping information by treating each I/O request as an *extent* and by using extents as basic mapping units rather than pages or blocks. By storing extents instead of individual addresses, our extent mapping schemes consume much less memory to store mapping information and still remain as flexible as page mapping. We observed in our experiments that our schemes reduced memory consumption by up to an order of magnitude in comparison with the traditional mapping schemes for several real world workloads. Our extent mapping schemes also scaled well with increasing address spaces by synthetic workloads. Even though the asymptotic mapping cost of VET and EMT is higher than the $O(1)$ time of a page mapping scheme, the amount of increased overhead was almost negligible or low enough to be hidden by an accompanying I/O operation.

1. Introduction

Flash memory solid state drives (SSDs) have been increasingly used as an alternative to conventional hard disk drives. The lack of moving parts in the flash memory devices frees them from long latency and excessive power consumption, and allows for lightweight design and strong resistance to shock.

Since flash memory does not allow any data to be updated in place, most flash memory devices come with a software layer called Flash Translation Layer (FTL) that is responsible for logically emulating conventional disk abstraction [1].

Fig. 1 represents FTL and NAND flash memory architecture. As shown in Fig. 1, FTL in general guides a write request arriving from a host into an empty area in flash memory, and manages mapping information from a *logical* address recognized by the host to a *physical* address in a flash memory device. For FTL to perform this logical-to-physical (L2P) address translation, a mapping table needs to be maintained inside a flash memory device.

Contemporary flash memory devices commonly rely on an address mapping scheme that uses a *page* (often 4 KB) or a *block* (usually 256 KB) as a unit of mapping. In page mapping [2], the granularity is a page, and flexibility is the foremost advantage since the logical address of a page can be mapped to any physical location on a flash memory device. As the capacity of flash memory devices grows, however, page mapping requires them to provide large RAM capacity for maintaining the large mapping table. In the case of a 4 TB flash memory SSD, for example, the size of its mapping table can be as large as 4 GB.

In block mapping [3], the granularity is a block, and the size of a mapping table is much smaller because of the larger granularity of mapping. In the above example, the table size would be only as small as 64 MB. However, block mapping has a critical shortcoming. The physical location of a logical page is fixed to a certain page offset within a block. Updating even a single page may require an entire block containing the page to be copied to an empty block. This makes a pure block mapping scheme impractical for most realistic workloads.

Hybrid mapping schemes (e.g., FMAX [4], FAST [5], Superblock-FTL [6], LAST [7]) have been proposed to take advantage of the strengths of both page and block mapping strategies. These hybrid schemes use extra flash memory blocks as an over-provisioned space where recently updated pages are stored without being restricted by their block boundaries, so that the addresses of these pages are managed more flexibly by page mapping. This hybrid strategy helps reduce the size of a mapping table, but this is only

* Corresponding author. Tel.: +1 520 370 1865.
E-mail addresses: yksuh@cs.arizona.edu (Y.-K. Suh), bkmoon@snu.ac.kr (B. Moon), alon@cs.arizona.edu (A. Efrat), jinsookim@skku.edu (J.-S. Kim), swlee@skku.edu (S.-W. Lee).

1383-7621/$ - see front matter © 2014 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.sysarc.2014.01.002

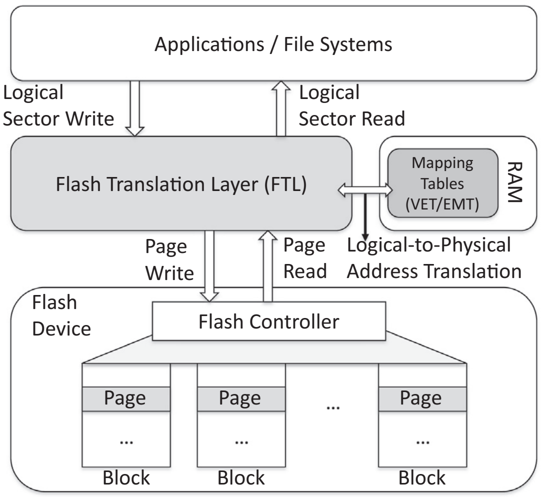


Fig. 1. FTL and NAND flash memory diagram.

feasible at the cost of over-provisioned flash memory (typically about 30% of usable capacity) and increased mapping latency.

In this article, we propose two novel mapping schemes that are fundamentally different from the existing mapping strategies. We call these new schemes *Virtual Extent Trie (VET)* and *Extent Mapping Tree (EMT)* (an enhanced version of VET), respectively, as they manage the mapping information such that a given I/O request is treated as an *extent* and the extent is used as the basic mapping unit [8]. By storing extents instead of individual addresses in requests, our extent mapping schemes consume much less memory space to store the mapping information and still remain as flexible as page mapping. Also, the proposed mapping schemes work regardless of underlying flash architecture by either single or multi-channels [9], since any type of physical address information returned after flash writes can be simply stored with the extents. As demonstrated in Fig. 1, the traditional mapping tables can be non-intrusively replaced by our extent mapping schemes. The preliminary results show that our extent mapping schemes can reduce memory usage by up to an order of magnitude compared with conventional mapping schemes for real world workloads. They also scaled well with increasing address spaces by synthetic workloads.

Though the asymptotic costs of mapping by VET and EMT are slightly higher than the $O(1)$ time of an address translation by page mapping, we expect that the amount of increased overhead would be almost negligible or low enough to be hidden by an accompanying I/O operation.

The article is organized in the following. Section 2 outlines the design of our extent mapping schemes. Section 3 proposes the algorithms for the VET and EMT schemes, and subsequently, Section 5 analyzes the complexities of the algorithms. Section 6 presents the performance evaluation results of VET and EMT using real and synthetic workloads. Section 7 provides related work. Lastly, Section 8 concludes this article.

2. Design principle

In this section, we discuss our design principle behind the VET and EMT schemes. We start by listing ideas that are shared by both schemes.

2.1. Extent-based mapping

In most traditional address mapping schemes, a page or a block is used as the unit of mapping. For a given read or write request from the host, FTL is responsible for translating all logical pages (or blocks) that the I/O request wishes to read or write to physical pages (or blocks) via its mapping table. Hence, FTLs must know which logical page (or block) is mapped to which physical page (or block) at all times by maintaining the mapping information in the mapping table.

The VET and EMT schemes we propose in this article takes advantage of the fact that an I/O request from the host consists of a logical start address and the number of sectors to read or write. Therefore, each I/O request can be considered an *extent* [8] defined in the logical address space and can be stored in the mapping table as a whole unit without being broken to multiple pages or blocks.

When a new write request arrives, it creates new mapping information or updates existing one. Specifically, if the request writes into a logical area which is not occupied by valid data, a new extent representing the write request will be created and inserted into the mapping table. The physical address information associated with the request will also be stored with the extent. If the request overwrites any valid data, one or more extents representing existing data will have to be updated. Those extents can be located by finding all extents that overlap the incoming one of the write request. A read request, in contrast, is treated as an *inquiry* extent. In order to translate the logical addresses of the read request to physical ones, our extent mapping schemes will look for all existing extents that overlap the extent of the read request and return physical addresses from the found ones.

Unlike a page or block mapping scheme whose granularity of mapping is fixed to either pages or blocks, the VET and EMT schemes store mapping information at the varying degree of granularity, which is determined solely by each individual write request. In other words, while page or block mapping handles only fixed-sized (one-to-one) mapping entries, our schemes manage variable-sized (many-to-one) mapping entries. Nevertheless, we observe the raw flash memory management (i.e., page or block allocation, garbage collection, wear-leveling regarding the limited number of erase counts, etc.) used in page mapping.

In subsequent sections, we describe what underlying structure is considered to design both schemes.

2.2. Virtual trie for VET

VET is a trie of binary strings but only in the logical sense (as will be discussed shortly). Hence, it is a *virtual* trie. Each binary string is composed of zeros, ones and special bits called *don't care* bits (denoted by '$*$' characters). The *don't care* bits can appear only at the end of a string. All strings have the same length but may have a different number of '$*$' bits. A binary string with a few trailing '$*$' bits in fact represents an extent whose length is a power of two. For example, an 8-bit binary string `0010****` can be used to represent an extent whose logical start address is `00100000` and whose length is 16. Since not every extent has a power-of-two length, an extent may have to be partitioned to one or more *canonical* extents before being added to VET. In other words, the mapping information of a write request is represented by one or more *canonical extents*. Formally, canonical extents are defined as follows.

Definition 1. An extent $\langle s, l\rangle$ is said to be canonical if the length l is a power of two and the start address s is a multiple of l.

A canonical extent $\langle s, l\rangle$ can always be represented by a single binary string, which can be obtained by replacing the least significant zeros in the binary representation of s with $\log_2 l$ '$*$' bits. For example, a canonical extent $\langle 8, 4\rangle$ can be represented by an 8-bit binary string (for simplicity) as below:

$$\langle \texttt{000010} * * \rangle.$$

In a virtual trie, a canonical extent serves as a *key* to identify each node.

Note that physical start address p associated with a canonical extent is excluded in Definition 1. Since p does not involve extent mapping in the article, for convenience of our discussion it is just omitted.

Fig. 2 shows a virtual trie to store the input extents in the canonical form as follows:

$$\begin{aligned} e1 &= \{\langle 0********\rangle, \langle 1000****\rangle\} \\ e2 &= \{\langle 11001***\rangle, \langle 11010***\rangle\} \\ e3 &= \{\langle 111*****\rangle\}. \end{aligned}$$

The VET scheme finds a set of canonical extents for the input extents, using node keys in top-down fashion, as shown in Fig. 2. For instance, given $e1$, the node a's key ($\langle********\rangle$) is compared with $e1$ in the virtual trie. Because the key is not equivalent to $e1$, VET determines which *child* of node a can subsume $e1$ in its entirety. In this case, neither of its children – nodes b and c – can completely contain $e1$; thus, $e1$ is split into smaller extents that retain candidate canonical ones. In this way, all the canonical extents of $e1$ can be found, and eventually $e1$ is kept by nodes b and g. Once the other input extents $e2$ and $e3$ are stored, the virtual trie will look like Fig. 2.

In a virtual trie, there exist two types of nodes: *leaf* and *internal* ones. They share the common key (or canonical extent) structure, but only a leaf node can have a given extent. The aforementioned leaf nodes b and g are linked to $e1$, leaf nodes k and n to $e2$, and leaf node o to $e3$, as shown in Fig. 2. An internal node, on the contrary, just serves as a *helper* that assists reaching leaf nodes. How the internal node gets exploited will be explained in Section 3.2.

A node in the virtual trie does not have any physical pointer to its child. Instead, a node is associated with its child nodes only logically and can identify them by unfolding the most significant *don't care* bit of the node to either '0' or '1'. For instance, nodes b and c in Fig. 2 can be determined as the left and right children of node a, as they share the same *don't care* bits except for the first one in their canonical extents.

2.2.1. Hash table

VET is a virtual trie, but it *physically* stores canonical extents into a *hash table*. The hash table is a traditional data structure to return the associated value with a hashed key via a hash function. In a virtual trie, a given extent is kept in the form of canonical ones that serve as node keys, thereby matching well with the hash table structure. Moreover, a hash table lookup does not rely on how many entries are in the table, as opposed to a physical trie structure. The average hash table lookup, therefore, is typically done within $O(1)$, assuming that significant overflows or collisions do not take place. These characteristics lead our decision to implementing a virtual trie by a hash table.

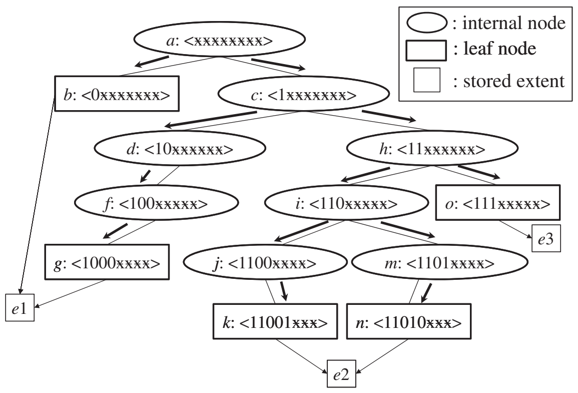


Fig. 2. Virtual trie by given extents $e1$, $e2$ and $e3$.

2.3. EMT

EMT is designed to enhance the performance of VET by more efficiently handling a write request. EMT can be implemented by any data structure that supports *find*, *insert*, *delete*, and *successor* or *predecessor*. EMT manages extents using the basic operations. In EMT, the *key* of a node is the logical start address of a write request representing an input extent. The *data* of that node comprises the *length* of the request and the corresponding *physical* start address, which is not explicitly represented in this article (as discussed in the VET scheme). We will discuss the algorithm of EMT in greater detail in Section 4.

3. Algorithms for virtual trie

An I/O request arriving from the host is either a read or a write operation that will be performed on a chunk of flash memory. Each operation comes with the logical start address and the size of a chunk. For a read operation, the task of VET is to translate the logical addresses to physical ones for the flash memory pages in the chunk. If the request is a write, VET is provided with the physical addresses where the data are actually written. VET is then responsible for updating the address mapping information. This section presents the key algorithms that VET uses to search for or update the address mapping information.

As described in Section 2, VET maintains the mapping information in the form of canonical extents and stores the extents in a virtual trie. Again, we implement the virtual trie using a hash table. Therefore, all the operations such as insertions, deletions and searches are explained in the context of a trie, but they are actually performed by hash insertions, deletions and lookups.

3.1. Update for write requests

A write request arriving from the host always causes data to be written to clean pages in flash memory and the address mapping information to be updated accordingly. If the data are written to a location whose logical address is not occupied by valid data, then a new piece of mapping information will be created and added. If the data are written to a location whose logical address overlaps – completely or partially – those of existing valid data, then the mapping information of the existing data being overwritten will be removed entirely or partially replaced by that of incoming data. Since we store the address mapping information in the form of canonical extents in the VET scheme, handling a write request will involve adding new extents to and removing old extents partially or entirely from a virtual trie. This section elaborates how these update operations are carried out in the VET scheme.

3.1.1. Inserting an extent

Each write request is treated as a given extent containing the start address and the size of data to be written. The first step toward processing a write request is to locate any existing extents overlapping the given one. This can be done by the search algorithm described in Section 3.2. The next task of the VET scheme is to reinsert the existing extents updated by the overlap (if any), deleting outdated extents, and finally to add the given extent.

In general, a given extent is not necessarily canonical as the request can be placed anywhere in the address space. Thus, it needs to be converted into one or more canonical extents. In the course of locating its canonical extents, VET not only creates all of its ancestor nodes but adds the canonical extent itself to the virtual trie. We call this scheme *LIS* (Linear Insertion Scheme).

To understand how LIS works, let us revisit Fig. 2 provided in Section 2. For $e3 = \langle 111*****\rangle$, for instance, the VET scheme using

LIS (linearly) inserts internal nodes *a*, *c*, and *h* followed by leaf node *o* pointing to *e*3. If any of the nodes already exists in the virtual trie, it is simply discarded. If leaf node *o* formerly had its old existing extent, it would be replaced by *e*3.

Algorithm 1. LIS

```
   input  : key (at the current level), e - A given extent
   output : NONE
 1 node ← Perform a lookup with key ;
 2 if key is a canonical extent of e then
 3     if node == NULL then Create a leaf node pointing to e.
 4     else
 5         if node == internal then  Delete the subtrie rooted from node.
 6         Have node point to e, removing the node's old extent if any.
 7     end
 8     return ; // Done.
 9 else
10     if node == NULL then  Create an internal node.
11     else if node == leaf then  node switches to internal.
12     Recursively find the rest of the canonical extents of e using the left or
       right child of key and splitting e if necessary.
13 end
```

Algorithm 1 represents LIS. In Line 1, VET performs a lookup for locating a node having a key. If the key is a canonical extent of a given extent *e* (in Line 2), we create a new leaf node with the key for *e* only if the lookup fails (in Line 3). If the lookup succeeds, then the VET scheme links the node found by the lookup to *e* (in Line 6). If the node is an internal, the VET scheme should remove the subtrie rooted from the node before the linking (in Line 5). (The deletion will be described in the next section.) Thereafter, LIS is terminated, as a leaf node is found.

If the key is not a canonical extent of *e* (in Line 9), then VET creates a new internal node if the lookup fails (in Line 10), and if the lookup succeeds locating a leaf node, VET switches the node to an internal one (in Line 11). (Certainly, if the found node is internal, then nothing is done.) After that, VET recursively searches for the rest of canonical extents of *e* by passing the left or right child of the current key and splitting *e* if *e* is partially subsumed by either of the children (in Line 12).

We can optimize LIS by taking advantage of how the extent search to be described in Section 3.2 is performed. We will discuss the optimization in Section 3.3.

3.1.2. Deleting an extent

If a given extent by a write request overlaps extents existing in the virtual trie, then the mapping information stored in the existing extent will be invalidated either completely or partially. If it need be invalidated completely, its corresponding extent will be removed from the virtual trie. If it need be invalidated partially, the corresponding extent will be divided so that only the invalidated portion can be removed from the virtual trie.

Fig. 3 exemplifies the partial invalidation. Suppose that a given extent $e4 = \langle \mathtt{1100100}* \rangle$ arrives at the trie. Because *e*4 overlaps *e*2 existing in the trie, *e*2 is decomposed into the following extents:

$$\begin{aligned} e4 &: \quad \{\langle \mathtt{1100100}* \rangle\} \\ e5 &: \quad \{\langle \mathtt{1100101}* \rangle, \langle \mathtt{110011}** \rangle, \langle \mathtt{11010}*** \rangle\}. \end{aligned}$$

By the partial invalidation, the incoming extent *e*4 overwrites one of the canonical extents of *e*2. Thus, the VET scheme creates internal node *p* and leaf node *q* for *e*4. It also switches (leaf) node *k* [1] (which used to keep one of the canonical extents ($\langle \mathtt{11001}*** \rangle$) of *e*2) to an internal one since it is no longer a leaf node for *e*2 being invalidated. In addition, now that the new extent *e*5 comprises the other three canonical extents, VET generates leaf nodes *r* and *t*, simply keeps leaf node *n*, linking it to *e*5, and finally evicts *e*2.

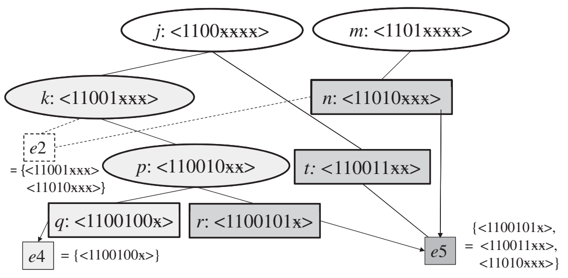


Fig. 3. Example of partial invalidation.

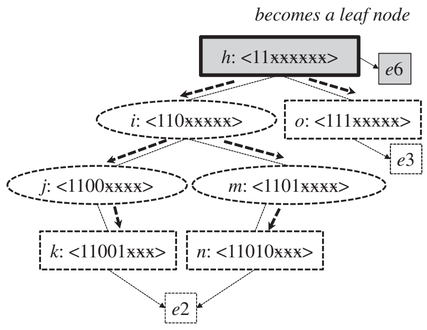


Fig. 4. Subtrie deletion.

Note that an incoming extent may require invalidating one or more existing ones at the same time. As illustrated in Fig. 4, the existing extents $e2 = \{\langle \mathtt{11001}*** \rangle, \langle \mathtt{11010}*** \rangle\}$ and $e3 = \langle \mathtt{111}***** \rangle$ get invalidated given an incoming extent $e6 = \langle \mathtt{11}****** \rangle$. This operation can be done in the way of locating and deleting all internal and leaf nodes used to store *e*2 and *e*3. It can also be optimized by eliminating an entire subtrie rooted from the previously common, internal node *h* for *e*2 and *e*3, followed by adding *e*6 to the node *h* now becoming a leaf one (as seen at lines 5–6 in Algorithm 1).

3.2. Search for read requests

When a read request arrives from the host, its logical address has to be translated to a physical one. This is done by searching the virtual trie for an extent containing the logical address. Since again the trie storing the mapping information is *virtual* and implemented by a hash table, the lookup operation can be carried out without traversing the virtual trie. Instead, it will be done more efficiently by performing a *binary search* against the nodes on a path (or *level*) of the virtual trie that contains the target node.

For example, if the logical address contained in a given read request is `204` (or `11001100` in binary representation), then the search begins at the mid point of the root-to-bottom path by taking only the first half of the binary string as a search key. In other words, the second half of the binary string is replaced by '$*$' (*don't care*) bits, and the first search key is formed as follows.

$$\langle \mathtt{1100}**** \rangle$$

If a lookup with the search key above succeeds and the match is found in a leaf node, then the search procedure will terminate. From the fact that the logical address `11001100` is contained in a canonical extent $\langle \mathtt{1100}**** \rangle$, the logical-to-physical address translation of `11001100` can be obtained immediately from the input extent of the leaf node.

[1] This node would be removed if we apply our optimization technique to be discussed shortly in Section 3.3, as it would not be used for locating either *e*4 or *e*5.

If the lookup succeeds but the match is found in an internal node, then the search procedure will continue on the lower half of the path. This is because the target node may still be found in the subtrie rooted by the internal node where the match is found. To continue the search procedure in the lower half of the path, a new search key is formed by restoring the first half of the bits masked off by '$*$' bits in the previous step. In this case of the example, the next search key will be

$$\langle \texttt{110011} * * \rangle.$$

A lookup may fail if there is no subtrie that contains the target node. However, that does not necessarily mean that there exists no canonical extent that contains the logical address. It could just mean that the binary search has gone down too far. If an extent is found at a level higher (than the non-existent subtrie), then the logical-to-physical address translation can still be obtained using the information stored by the extent. Therefore, the search must continue by narrowing down the scope upwards on the path (or by adding more trailing '$*$' bits in the search key). Continuing the running example, the next search key will be

$$\langle \texttt{11001} * ** \rangle.$$

In a nutshell, the direction of the next search is determined by the existence of an internal node storing the current search key. Fig. 5 depicts the search process along the path of the virtual trie.

Note that the scenario given in the example above is somewhat oversimplified. If a read request is large, then its extent may not be covered entirely by a single canonical extent found in a leaf node. In this case, the search must continue by forming a new search key for the unresolved portion of the logical address (excluding the extent partially covered by the canonical extent).

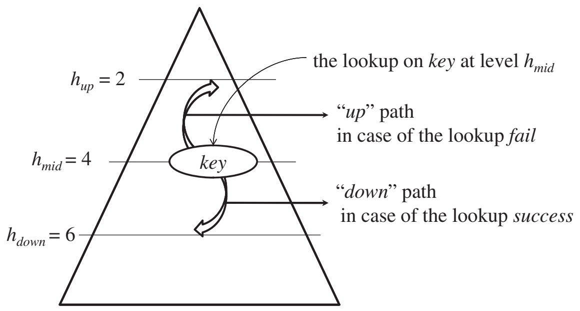
Algorithm 2 elaborates the search procedure described above. Note that a stack of storing search keys is leveraged, and again the direction of the search is determined by the result of a lookup with a popped key.

Algorithm 2. Extent search.

```
input   : e - A given extent
output  : E - A list of existing extents overlapping e
1  Push the very first search key into stack.
2  while stack is not empty do
3      node ← Perform a lookup with a popped key.
4      if node ≠ NULL then                                   // Lookup success
5          if node == leaf then                               // Leaf node
6              Put the existing extent kept by node into E, updating the new
               start address using the found extent.
7              Start over the search with a new key if possible.
8          else                                               // Internal node
9              Continue the search on the lower half of the path with a new
               key if possible.
10         end
11     else                                                   // Lookup failure
12         Continue the search on the upper half of the path with a new key if
           possible.
13     end
14 end
```

3.3. Optimization by binary insertion scheme

A careful observation by the aforementioned binary search could lead to the improvements on the insertion procedure of LIS in terms of memory usage and processing time. Namely, we can see that some ancestors for a canonical extent are not used for the binary search on the target node. For instance, leaf node k in Fig. 2 has its following ancestors:



the lookup on key at level h_{mid}
$h_{up} = 2$
"up" path in case of the lookup fail
$h_{mid} = 4$
key
"down" path in case of the lookup success
$h_{down} = 6$

Fig. 5. Binary search on VET levels.

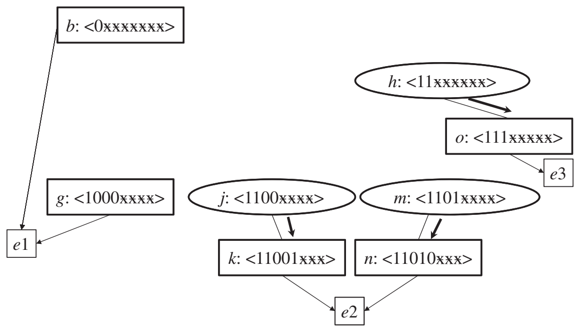


Fig. 6. Virtual trie via BIS by given extents $e1$, $e2$ and $e3$.

internal node$a$	$* * * * * * * * *$
internal node$c$	$\texttt{1} * * * * * **$
internal node$h$	$\texttt{11} * * * * * *$
internal node$i$	$\texttt{110} * * * **$
internal node$j$	$\texttt{1100} * * * *$

Node j is the first (and only) one to become visited in the process of the binary search against the target leaf node k. In other words, internal nodes a, c, h and i will not be used in the rest of the search. Hence, it is unwise to insert those internal nodes except node j for locating leaf node k. By adding only a vital internal node (s), we are able to not only spend less time on but also save much memory for the insertion of an extent. This optimization scheme is called *BIS* (Binary Insertion Scheme).

Fig. 6 represents the virtual trie made by BIS. Most of the internal nodes are not seen, as opposed to Fig. 2. It is because the BIS approach derives internal nodes only lying along with the binary search path for locating each canonical extent, and adds them to the virtual trie.

Algorithm 3. BIS

```
input   : key (at the current level), e - A given extent
output  : NONE
1  if key is a canonical extent of e then
2      Locate and delete an invalid extent if any.
3      Insert internal and leaf nodes, considering the binary search path on
       key.
4      Delete the subtrie rooted from the leaf node having key.
5      return ; // Done.
6  else Do line 9-12 in Algorithm 1.
```

Algorithm 3 shortly describes the process of BIS. Most of the parts are similar to those of Algorithm 1. The BIS algorithm, however, is slightly different, since it should locate and remove a leaf node with an invalid extent that may lie above a newly inserted internal node, as illustrated at line 2. For instance, in Fig. 3 node *k* with *e*2 must be evicted before the insertion of internal node *p*. This extra operation is not needed by LIS because node *k* will have been deleted already by the time the internal node *p* gets inserted.

4. Algorithms for EMT

In VET, it is not cheap to accommodate a new extent into a virtual trie. The VET scheme should locate any extents overlapping with and create nodes for the new extent while eliminating nodes containing the invalid extent and adding back the valid extents if any. In particular, small writes incur inserting more nodes into and spending more time on existing extent search and node deletion in the virtual trie.

In this section, we discuss EMT, which is an enhanced version of the VET scheme. The EMT scheme mainly improves the write processing time and memory usage, compared with VET. We can implement EMT using any abstract data structure having *search*, *insert*, *delete*, and *successor* or *predecessor* operations.

The two key technical differences between EMT and VET are in the following. First, each node in EMT should determine whether the start address of a given I/O request is contained in the stored extent that can be restored from the node's key and data. Recall that the logical start address of an I/O request is used as *node key*, and the length of the request and its physical start address used as *node data*. If a node has the start address, then the node is eliminated from EMT in the case of a write request (due to overlapping) while it is included in the answer set in the case of a read request. The second distinction is to take advantage of *successor* and *predecessor* operations of the abstract data structure. The operations are used to locate the next node when the existing extent stored by a current node partially subsumes the given extent. In the following sections, we will specifically describe how a given write or read is served in EMT.

4.1. Update for write requests

The update by a given write request can be similarly treated as is done by the VET scheme. Namely, EMT (1) first locates and deletes all overlapping extents with the write, and (2) subsequently inserts the new extent and any non-overlapping (non-invalidated) ones at a time if any. After detecting an extent partially or entirely superseded by the given write, the EMT scheme leverages the predecessor or successor operation of a tree, to continue to retrieve any remaining extents. (On the contrary, recall that VET resumes the search by re-generating a lookup key through the update of the next start address.)

Fig. 7 represents an example of how a new extent insertion is handled in the EMT scheme. Fig. 7(a) shows a logical view of EMT, assuming that *e*1, *e*2, and *e*3 seen in Fig. 2.2 are already stored, and a new extent $e7 = \langle 80, 160 \rangle$ arrives. The EMT scheme first locates *e*1 based on the start address of *e*7. As *e*1 partially overlies, the valid extent of *e*1 (*e*1′) is left (re-inserted), and the invalid is replaced by *e*7. Then, EMT finds *e*2 entirely covered by *e*7, thereby evicting *e*2. Lastly, the EMT scheme partially invalidates *e*3, as done on *e*1, leaving the valid (*e*3′). Fig. 7(b) represents the physical nodes of EMT before and after *e*7 is treated. Note that each node represents an extent as opposed to VET consisting of internal and leaf nodes, at which stores the extent.

Algorithm 4. Processing a write request in EMT

```
   input  : $e_n = <s_n, l_n>$ (A given extent by a write), $T$ (EMT)
   output : None
 1 $I = \{e_n\}$ ;                              // I = A set of extents to be added
 2 $e_t = T.predecessor(s_n+0.001)$ ;                       // $e_t$ = <$s_t$, $l_t$>
 3 while $e_t \neq NULL$ do
 4     if $s_t < s_n$ and $s_n < s_t+l_t-1$ then
 5         $I = I \cup \{<s_t, s_n-s_t+1>\}$ ;       // Leave the non-overlapping of $e_t$
 6     end
 7     if $s_t < s_n+l_n-1$ and $s_n+l_n-1 < s_t+l_t-1$ then
 8         $I = I \cup \{<s_n+l_n, s_t+l_t-s_n-l_n>\}$ ;  // Leave the non-overlapping of $e_t$
 9     end
10     $e_{t'} = e_t$ ;
11     $e_t \leftarrow T.successor(s_t)$ ;
12     $T$.delete($s_{t'}$) ;  // Now, invalidate $e_{t'}$ (that is the previous $e_t$)
13     if $s_n+l_n-1 < s_t$ then break ;    // No more overlapping extents
14 end
15 $T$.insert($I$) ;
```

Algorithm 4 specifically describes the extent insertion in the EMT scheme. Note that the insertion may accompany extent deletion and update. First, EMT searches for the predecessor extent of a given start address (s_n) (line 2). The reason to add a slight margin (e.g., *0.001*) to s_n is to catch the case that s_n happens to be the same as the start address of an existing extent. If the start address (s_t) of e_t is less than s_n or the end address of e_n is less than that of e_t (lines 4 or 7) (partial invalidation), then EMT adds only the valid portion of e_t while replace the overlapping of e_t by e_n (lines 5 or 8). After that, EMT updates e_t to its successor for the next iteration (line 11), and deletes $e_{t'}$ (e_t before the update) to complete the invalidation (line 12). If the last address of e_n is covered by e_t (line 13), then EMT terminates the insertion of e_n by adding every extent in *I* (line 15).

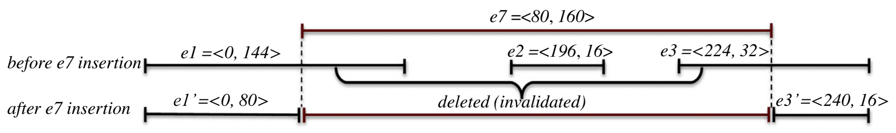
4.2. Search for read requests

As discussed in VET, a read request is translated as a query extent in extent mapping. Serving the query extent in EMT can be similarly done as locating overlapping extents with a given write, but it is slightly different than that of the write in that *page fault* may occur once EMT knows no node containing any address of the read.

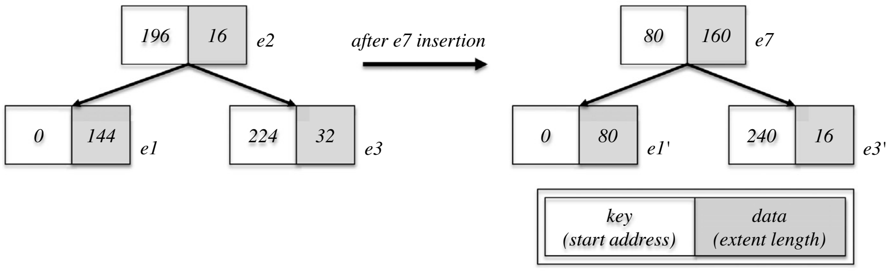
Algorithm 5. Extent search in EMT.

```
   input  : $e_q = <s_q, l_q>$ (query extent), $T$ (EMT)
   output : $E$ - A list of extents overlapping $e$
 1 $p \leftarrow s_q$ ;                                  // Search start address
 2 $e_t = T.predecessor(p+0.001)$ ;                         // $e_t$ = <$s_t$, $l_t$>
 3 while $e_t \neq NULL$ do
 4     if $p \in e_t$ then
 5         $E = E \cup e_t$ ;
 6         if $(s_q+l_q-1) > (s_t+l_t-1)$ then // $e_q$'s end address not covered
 7             $p = s_t+l_t$ ; // Update the next search address beyond $e_t$
 8             $e_t = T.successor(s_t)$;
 9         else return $E$;                                          // Done
10     else break
11 end
12 return $NULL$ ;                                           // Page fault
```

Algorithm 5 describes the extent search on a given extent (e_q) by a read arriving at EMT. First, EMT looks for a node (e_t) that contains the start address ($s_q = p$) of e_q. If e_t exists, we add it to *E*, which keeps a list of extents overlapping e_q (line 5). If e_t does not



(a) Logical EMT before and after $e7$ insertion



(b) Physical EMT before and after $e7$ insertion

Fig. 7. A new extent insertion in EMT.

cover the last address of e_q (line 6), then p is updated to the next address after e_t, and the succesor of e_t is retrieved with $O(1)$ time for the next comparison (lines 7–8). Otherwise, the extent search for e_q is terminated by returning E. If e_t is absent or p is not contained in e_t (line 10), EMT sees e_q as *page fault* of reading not-yet-written flash pages (as no stored extents contain the given address) and terminates the search with no discovered extent (line 12).

5. Analysis

We study the complexity of the proposed algorithms.

5.1. VET

In this section, we provide asymptotic worst case space and time analysis of the VET scheme.

5.2. Generated nodes

Lemma 5.1. *During the LIS algorithm, the number of new nodes generated for each new extent is* $O(\log|U|)$. *This number drops to* $O(\log\log|U|)$ *if BIS is used.*

Proof. Suppose that we insert the shortest extent in the $O(\log|U|)$-address space under LIS. To insert this extent, we need $(\log|U| - 1)$ internal nodes and one leaf node that stores that extent. Therefore, the number of new nodes for storing that extent is $O(\log|U|)$ in LIS. Under BIS, in the worst case VET generates $O(\log\log|U|)$ internal nodes and one leaf node pointing to the extent. Hence, the number of new nodes generated by BIS is $O(\log\log|U|)$. □

5.2.1. Update for write requests

As discuss in Section 3.1, the update for write requests incurs (1) existing extent search followed by (2) extent insertion by LIS or BIS accompanying extent deletion.

Lemma 5.2. *The running time complexity for the update by a given extent e is* $O((k+1)\log|U|)$*., where k is the number of the existing canonical extents overlapping e.*

Proof. The proof follows similar arguments for segment trees [10]. Assume that k existing canonical extents overlap the given extent e. First, it takes $O(k\cdot\log\log|U|)$ time to locate all the canonical extents, since each can be found within $O(\log\log|U|)$ time (as will be proved in Lemma 5.3). Next, VET can determine and insert each canonical extent of e in $O(\log|U|)$ time. Lastly, the VET scheme by LIS (by BIS) needs to eliminate $O(\log|U|)$ (resp. $O(\log\log|U|)$) ancestor nodes associated with each overlapping canonical extent.

In short, the total running time complexity for the update is equal to $O((k+1)\log|U|)$ □

In practice, our experiments showed that k was less than 1.45 on average (as will be demonstrated in Fig. 12).

5.2.2. Search for read requests

Lemma 5.3. *VET answers a query extent in time* $O(k\cdot\log\log|U|)$, *where k is the number of the existing canonical extents that overlap the query extent.*

Proof. Recall that VET carries out the binary search on the path for locating the canonical extent containing a start address of e. Because the length of the path is $O(\log|U|)$, the search on the address can be done within $O(\log\log|U|)$ time. Suppose that a virtual trie has a total of k existing canonical extents overlapping e. Then, the search is performed k times. Therefore, the total time complexity for the search is $O(k\cdot\log\log|U|)$. □

In our experiments, k was on average no greater than *1.19* (as will be shown in Fig. 12).

5.2.3. Canonical extents

Lemma 5.4. *At most* $2\log|U|$ *canonical extents can be inserted for every extent.*

Proof. Note that while inserting a new extent e, we create a canonical extent e_c iff

$$\begin{aligned} e_c &\subseteq e \\ e_d &\not\subseteq e,\ \text{where } e_d \text{ is a child of } e_c. \end{aligned}$$

Hence, at most two nodes are created at each level of a virtual trie. □

The bound of $2\log|U|$ was over-pessimistic, and our experiments indicate it is on average no bigger than 2.79 (as shown in Fig. 12).

5.3. Asymptotic performance analysis of EMT

Consider a new extent e overlapping a set of previous extents. (See $e7$ in Fig. 7(b)). First consider the ones containing the endpoints of e. These are one or two extents containing these endpoints, and these extent (s) are modified, and possibly one new extent is created. The other (zero or more) extents that e overlaps (e.g., $e2$ in Fig. 7(b)) are fully contained within e, and hence are deleted after e is inserted. Since an extent could be created only once, we obtain

Lemma 5.5. *Consider any sequence of insertion of m extents into an empty data structure. Then during this process at most $O(m)$ extents are being deleted, and $O(m)$ are being modified.*

To support these operations in asymptotically efficiency, one could use the y-fast trie [11]. Similar to the famous van Emde Boas trees [12], these trees assume that keys are taken from a universe of integers of cardinality U, and support *Find*(x), Successor (x) and *insert/delete*, each in time $O(\log\log U)$. However, they improve the van Emde Boas trees by requiring only linear space (in the number of created extents). Putting it together with Lemma 5.5, we obtain that

Lemma 5.6. *If the y-fast trie is used, then the time for performing any sequence m insertion of extents is $O(m\log\log U)$, so the amortized time for each insertion is $O(\log\log U)$. This is also the time for a query (extent).*

Due to availability of code, and somehow the involved structure of y-fast trie, we have opted to use a standard balanced binary search tree instead in our implementation. The asymptotic time in this case is slightly higher than specified in Lemma 5.6. Since the time for each operation is $O(\log m)$, we obtain from Lemma 5.5.

Lemma 5.7. *If a balanced binary search tree is used, then the time for performing any sequence m insertion of extents is $O(m\log m)$, and thus, the amortized time for each insertion is $O(\log m)$. This is also the (worst case) time for a query.*

6. Experiments

In this section, we describe the environment settings for our experiments. Next, we evaluate and analyze the performance of our extent mapping schemes – VET and EMT – using real-world and synthetic workloads, compared with the traditional mapping schemes.

6.1. Environment settings

We implemented all the algorithms in C language. The most ideal, accurate configuration for the evaluation of the algorithms would be to use a real ARM processor in flash memory devices. Unfortunately, it was not possible to obtain the processor. To make our evaluation more realistic, we used a low-end 32-bit machine with a Pentium 4 CPU 3.00 GHz processor and 2 GB memory, running the Linux system with a 2.6.32–32-generic kernel version.

Table 1 describes the workloads used for our experiments. The real world traces – `finance`, `homes`, `wdev` – were obtained from public I/O trace repositories [13,15], and `wsf` was the real workload extracted from the web activity. The `Spew` trace gained by a workload generator [16] comprises only write requests. As discussed in Section 6.3.2, this trace was specially designed for the memory overhead analysis with an increasing address space.

In addition, we provide the histograms about request address and size in the real workloads, as illustrated in Figs. 8 and 9. The statistics about the requests gives us an important basis in analyzing the performance of our extent mapping schemes along with the workloads.

6.2. Extent mapping performance evaluation

This section provides the performance evaluation results of our extent mapping schemes.

6.2.1. VET

We give an analysis of the overall performance of VET on the real world traces, with respect to memory usage, elapsed times, and canonical extents, and compare the performance of the insertion schemes (LIS and BIS) used in VET.

Memory consumption. Table 2 provides the memory footprint of VET over the real world traces. The amount of memory needed to treat the traces was relatively very little when considering their address spaces (although there was some mapping overhead observed in Table 3). Specifically, the VET scheme only used at most about 17 MB memory for processing the `finance` trace, which had the largest address space, irrespective of whichever insertion methods (LIS and BIS) was employed. Apparently, `homes` consumed the greatest amount of memory among the traces. It was because the `homes` workload revealed the most widespread, unused range of request addresses, as illustrated in Fig. 8, and it had far more write requests than any other workload, as indicated in Table 1. This implies that a large amount of mapping information needed to be created across the entire address space, and correspondingly, much memory was needed for `homes`. In contrast, the write requests of `wdev` were likely to show a relatively narrow address range compared with the other traces. In addition, the same request addresses in the trace tended to be accessed repeat-

Table 1
Real/synthetic workloads used for the performance evaluation.

Type	Name	Description	Address space	# of Writes Average write size	# of Reads Average read size
Real	`finance`	OLTP application [13]	644 GB	8.2 M 3.7 KB	2.5 M 2.5 KB
	`homes`	MS exchange servers [14,15]	542 GB	16 M 4 KB	0.4 M 12.0 KB
	`wdev`	MS exchange servers [14,15]	51 GB	1.1 M 8.2 KB	0.23 M 12.6 KB
	`wsf`	Web surfing activity on PC	32 GB	0.3 M 22.5 KB	93 K 19.8 KB
Synthetic	`Spew`	`Spew` [16] workload generator	16 GB∼1 TB	50 K 83.8 KB∼376.8 KB	0 N/A

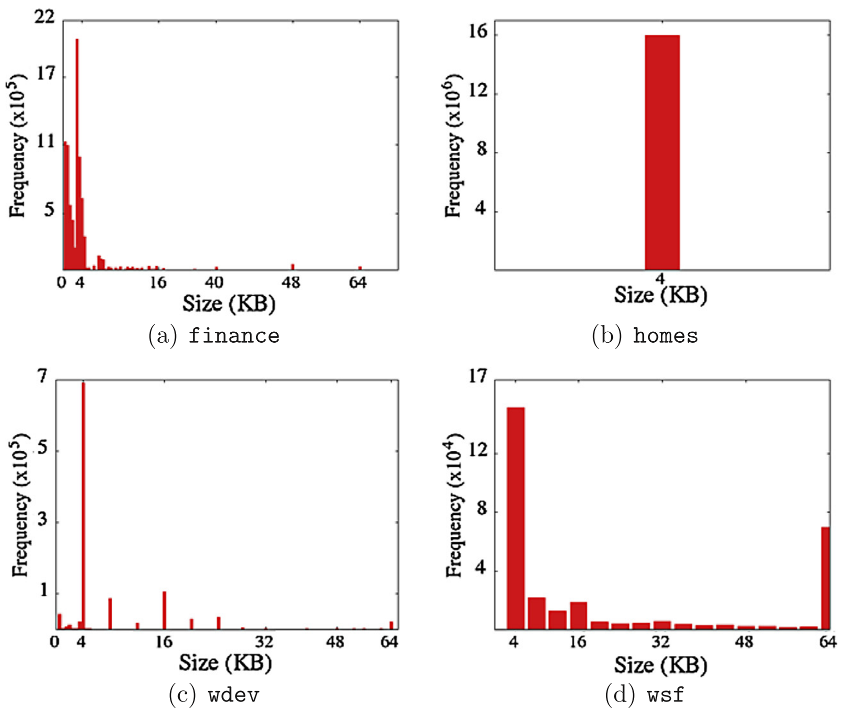


Fig. 8. Histograms on write request address.

edly, thereby incurring no extra mapping information written. For this reason, the wdev trace needed less memory than the others. (The VET scheme is definitely advantageous to this type of workload.) The finance workload appeared to have as narrow an address range as that of wdev. However, many of the addresses not clearly seen in Fig. 8(a) were far more widely scattered within and even outside the visible address range. Of course, the number of the requests was by far greater than that of wdev as well. The wsf trace seemingly had a broader address range than the finance one, but actually, it used only a few addresses in the entire range, and consequently, the memory consumed by the VET scheme was much less than that of finance.

Elapsed time. Table 3 provides the average elapsed time on a request of the real world traces. Typically, the average elapsed time on a write request was greater than that of a read one. That was because the write request incurred additional VET operations such as extent insertion or deletion besides common existing extent search. The mapping time on the write request (~25 μs), however, can be hidden or low enough by following flash writes. Hence, the mapping cost is regarded sufficiently negligible.

Fig. 10 exhibits the full distribution of the elapsed times on write requests of the traces. Most write requests were processed within a couple of microseconds, other than some taking very few taking hundreds of milliseconds. Note that most of the points are gathered right above the X-axis. A remarkable variation of the times were not observed across the workloads.

Fig. 11 shows all the elapsed times on read requests of the traces. There seemed some requests taking a few hundreds microseconds, but VET minimized the mapping time within an average of at most around 2.5 μs. We believe that the mapping cost rarely hurts the flash read performance (even if VET is applied on the real ARM processor). Also, the times did not vary substantially as seen in Fig. 11. The results testify that our binary search technique work very effectively.

One thing to notice is that the elapsed times on I/O requests differed by workloads. (Of course, the total memory for each trace varies too.) The finance trace revealed the highest average elapsed time among the traces. The primary reason was involved with canonical extents associated with I/O requests. We will discuss more details about this in the following.

Canonical extents. We study the correlation between canonical extents and the performance of the VET scheme. Fig. 12 illustrates the average canonical extents in regard to an I/O request. For each workload, the first and second bars mark average canonical extents that were generated by and overlapped a given extent (by a write), respectively. The third one indicates the average canonical extents returned to an inquiry extent (by a read).

On the whole, finance had the most average canonical extents involving an I/O request. Meanwhile, the homes trace had the fewest average canonical extents produced by each request. Concerning the other traces, the write requests of wdev encountered slightly more overlapping canonical extents but generated only about 50% fewer canonical extents than those of wsf on average. The read requests of wdev were responded with much fewer average canonical extents than those of wsf as well.

The observations above fairly coincided with the average elapsed time results shown in Table 2. Given an I/O request, the fewer canonical extents were involved, the faster response times (as well as the less memory) were observed.

Note that on average at most about three canonical extents were generated per request of the traces, as indicated by the first bars in Fig. 12. This demonstrates that the bound of Lemma 5.4 is overly pessimistic, thereby not being a serious concern in practice with respect to memory overhead.

To summarize, canonical extents involving a workload can be important factors in the performance of VET.

Comparison between LIS and BIS. The optimization by BIS in the VET scheme was successful, as already exhibited in Tables 2 and 3. Across the real traces, virtual tries built by BIS typically outperformed those by LIS in terms of memory usage and processing times. Again, when applying BIS, we can insert the fewer number of internal nodes, thereby consuming less memory. In particular, the VET scheme using BIS overwhelmed the one using LIS by about a factor of two when processing write requests. Moreover, adding

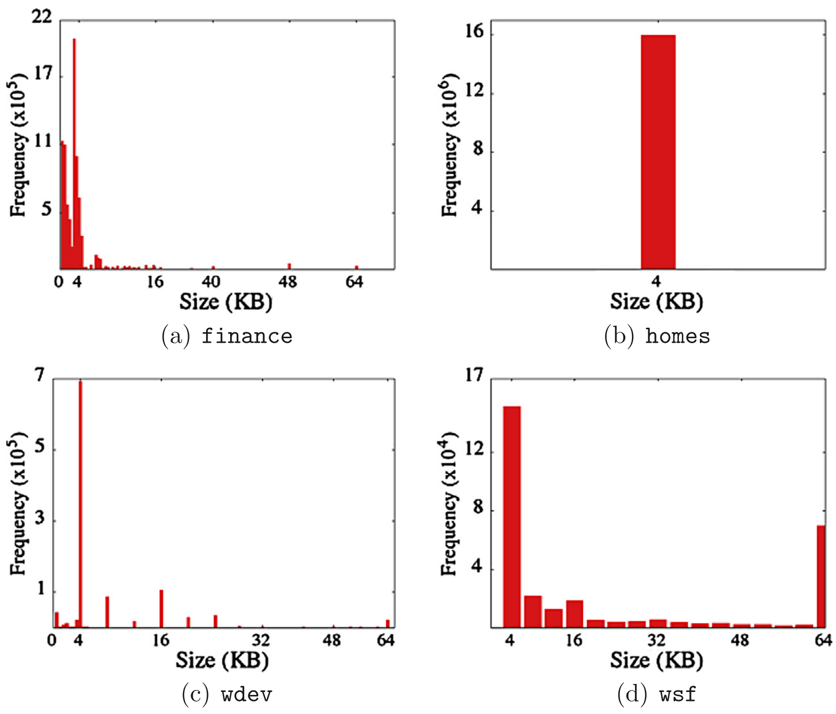


Fig. 9. Histograms on write request size.

Table 2
Memory footprint.

	Type	finance	homes	wdev	wsf
Memory usage (MB)	BIS	15.07	23.25	1.15	4.69
	LIS	16.89	28.50	1.63	6.03

Table 3
Average elapsed times.

	Type	finance		homes		wdev		wsf	
		Write	Read	Write	Read	Write	Read	Write	Read
Avg. elapsed time (us)	BIS	16.84	2.17	4.43	0.99	10.29	0.95	9.64	1.44
	LIS	24.63	2.47	8.25	1.07	9.44	1.00	12.97	1.53

fewer nodes may be beneficial to locating any extent (s) faster in a hash table (e.g., myhash [17] to be discussed later in the following section) such that more than two (internal or leaf) nodes having an identical hash key may co-locate by chaining in the same bucket. Consequently, it took less average elapsed times to serve a read request across the workloads.

To summarize, we prefer BIS to LIS for the VET scheme. Consequently, we will show the results only by BIS in the rest of experiments.

Hash table choice. The VET scheme provides the flexibility of employing any hash table implementation. Although VET is not limited to a specific hash table, it is interesting to see which hashing scheme fits well to the VET scheme, yielding the smallest performance overhead. Thus, in this section, we study how different VET performance can be seen in accordance with hash table implementations.

Among quite a few hash tables in literature, for the VET scheme we considered the following three candidates: cuckoo-hash [18], uthash [19] and myhash [17]. Due to space constraint, we omit the description of each of the hash tables. (For more details, see Refs. [18,19,17].)

Table 4 presents the performance comparison results among each hash table implementation. It was myhash that presented the most efficient performance for VET. Specifically, when treating write requests, myhash not only used 16% and 58% less memory but revealed 18% and 43% faster elapsed times than cuckoo-hash and uthash, respectively. In addition, myhash achieved average speedups of up to about 18% and 30% when serving read requests, compared with cuckoo-hash and uthash, respectively. That was mainly because the myhash's hashing function worked well for node keys used for the VET scheme, and its chaining technique for handling bucket overflow also consumed less memory than the others. All the experimental results presented before this subsection, therefore, are based on myhash. The worst one was uthash. It required a 24 KB-handle per node for making it hashable and carrying a variety of accessory features actually unnecessary for VET. Therefore, it was not relatively fast and lightweight. The cuckoo-hash implementation showed fairly as comparable

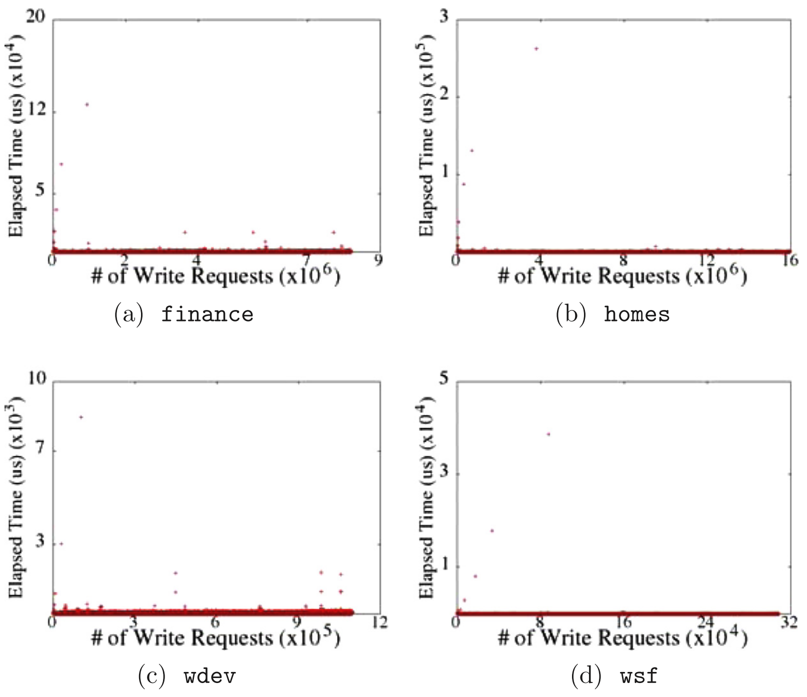


Fig. 10. Elapsed time distributions – write request.

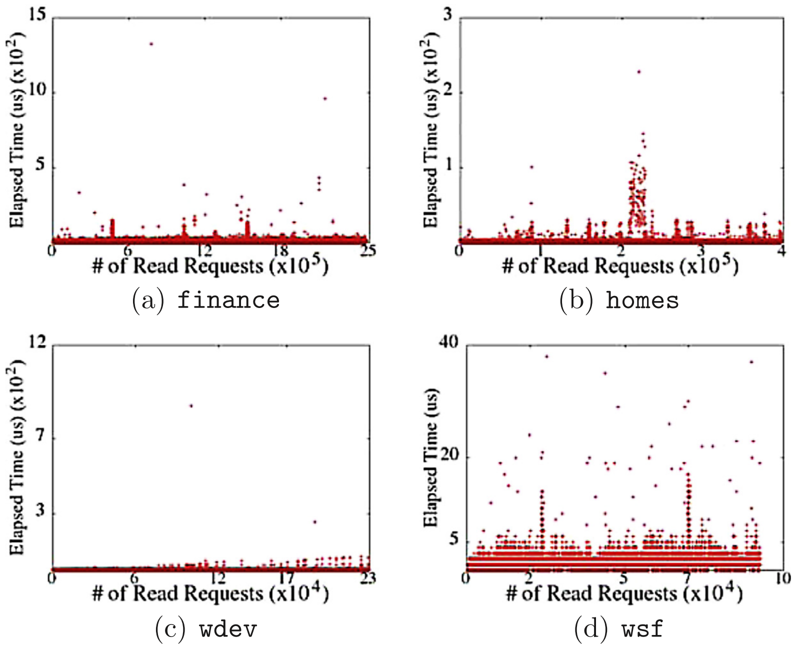


Fig. 11. Elapsed time distributions – read request.

performance as myhash, but it needed more memory due to a few extra bytes added per node for linear probing to cope with the overflow.

In conclusion, the performance of the VET scheme may somewhat vary among hash table implementations, and therefore, a careful choice needs to be made.

Cache-aware implementation. The VET scheme that has been explained so far assumes that an input extent is pointed to by a leaf node in a virtual trie. It is called the *Pointer-Dependent* VET (PD-VET) scheme. In the PD-VET scheme, the use of an external pointer at a leaf node to such a given extent could be a concern in performance for the following reasons. First, when VET creates a leaf node, extra memory allocation should be required for storing an input extent attached to the leaf. Second, whenever a stored extent is accessed at a leaf node, PD-VET suffers from *pointer-chasing*, which may aggravate *cache-hit* ratio. To avoid the overhead by the pointer use, as an alternative we can consider the *Pointer-Free* VET (PF-VET) scheme.

Keeping the same key (or canonical extent) structure, PF-VET allows only leaf nodes to directly embrace an input extent (with no pointer). In other words, internal and leaf nodes can be distinguished by the value of *extent length* field. Specifically, when the lookup using a given key succeeds, the returned node is identified as a *leaf* if its length value is positive. Otherwise, it is regarded as an

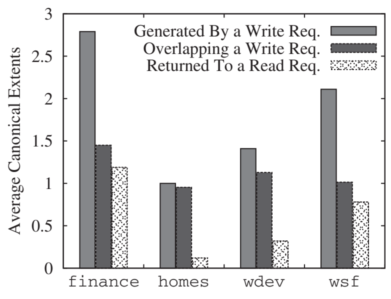


Fig. 12. Canonical extent statistics on a given I/O request.

internal node. By doing away with the *pointer-dependency* simply using the length field, the PF-VET scheme is expected to alleviate not only memory allocation overhead but also reduce *cache-misses*, thus eventually enhancing the overall performance of a virtual trie.

For our experiments, we reorganized the aforementioned real workloads in a way that all write requests in the real traces came before read ones to obtain the exclusive cache misses along with request type in both of the VET implementation design choices. To capture the misses, we leveraged the *Performance Application Programming Interface* (PAPI) library [20]. Due to the PAPI installation issue, only this experiment was inevitably conducted on a 64-bit machine equipped with dual cores with 4 MB cache. We think that the machine change does not impact the fair evaluation between the two implementations.

The experiment results show that our cache-aware design is quite promising. As shown in Fig. 13, overall the PF-VET slightly reduced the average cache misses in L1 or L2, compared with the PD-VET, thus leading to a slight speedup up to around 10% on the average elapsed times on write or read requests. In particular, the hits in L2 cache were increased even up to 15% on `finance`, as shown in Fig. 13(b). Note that L2 cache misses typically cost 20 times than L1 cache ones [21]. This can explain why PF-VET outperforms PD-VET in terms of the average elapsed times on write requests even though L1 cache misses in PF-VET on some traces were slightly higher than those in PD-VET. Our results not only raise the importance of avoiding pointers in implementation, but also addresses a correlation between cache misses and elapsed times.

6.2.2. Mapping time comparison between VET and EMT

In this section, we compare the extent mapping time overhead exposed in VET and EMT. Fig. 14 shows average elapsed time taken to serve a write or a read in both schemes. The times related to VET are borrowed from Table 3 for the comparison with the corresponding ones to EMT. As you can see, EMT obtained significant gains up to an order of magnitude in terms of the average write processing times, compared to VET. This implies that EMT can be a better choice than VET over general workloads. It is because that the former can finish overlapping extent searches or deletion at a time in an amortized way, thanks to a node pointer exploited by successor or predecessor operations. Note that extent deletion in VET could be aggravated on the workloads in which input extent lengths vary a lot, thereby considerably affecting the overall write elapsed time. On the contrary, the VET scheme slightly outperforms the EMT scheme when treating reads. This is because the extent search in VET does not does not depend on how many input extents are stored in a virtual trie, as opposed to the EMT scheme.

Table 4
Performance comparison over different hash tables.

Trace	Perf.	Hash tables		
		`cuckoo-h.`	`uthash`	`myhash`
`finance`	Mem.	18.06 MB	36.03 MB	15.07 MB
	Write	17.66us	19.05us	16.84us
	Read	1.78us	2.24us	2.17us
`homes`	Mem.	27.75 MB	54.73 MB	23.25 MB
	Write	4.93us	5.20us	4.43us
	Read	1.18us	1.43us	0.99us
`wdev`	Mem.	1.37 MB	2.74 MB	1.15 MB
	Write	12.59us	18.07us	10.29us
	Read	1.11us	1.36us	0.95us
`wsf`	Mem.	5.59 MB	10.99 MB	4.69 MB
	Write	10.85us	13.48us	9.64us
	Read	1.51us	2.00us	1.44us

6.3. Performance comparison with traditional mapping schemes

In this section, we compare memory consumption by traditional and extent mapping schemes over real and synthetic workloads.

6.3.1. Memory consumption over real traces

As shown in Fig. 15, the VET/EMT schemes used much less memory than a page mapping table (PMT) [2] or a hybrid mapping table (HMT) [4–7]. Specifically, when treating the workloads, VET/EMT consumed only about 2.3/0.7%, 4.3/3.7%, 2.3/0.9%, and 27.9/9.5% of the total memory required by the PMT in order. Even compared with the HMT, the VET/EMT schemes used only about 5.3/1.6%, 9.8/8.5%, 5.2/2.1% and 63.8/19.5% of the HMTs total memory in order. This is attributed to the fact that the memory consumption by our extent mapping is not dependent on address space size but mainly determined by *workload characteristics*. The characteristics can be defined in accordance with how small or big write requests are, and how scattered or narrow the request addresses are.

For instance, in spite of the huge address spaces more than 0.5 TB, the writes of the `finance` or `homes` traces touched only a few portions of each of the spaces. Hence, our extent mapping schemes did not need much mapping information for the requests, thereby being able to save a huge amount of memory.

The `wdev` trace had a small address space (51 GB), but again many of the writes tended to use the same addresses accessed before, thus inducing no additional memory consumption. In the case of `wsf`, it had the smallest address space (32 GB), and even fewer portions of the space were left intact unlike the other traces. Hence, the memory reduction percentage on `wsf` was not so great as that of the others. Nevertheless, no trace requires more memory in our schemes than in the conventional ones.

Lastly, EMT typically uses less memory than VET by up to about a factor of three. That is because VET trades with read mapping time additional memory for creating internal nodes (for guiding to leaf nodes). EMT also has internal nodes, which are different from those of VET in that every node in EMT can contain input extent.

All in all, our extent mapping schemes outperform the traditional mapping schemes in terms of memory consumption. The memory reduction by extent mapping schemes can be particularly maximized by the workload, such that one typically accesses only a few portions of a huge address space and has moderate request size.

6.3.2. Scalability comparison over increasing address space

We study how well our extent mapping schemes scale as the address space of flash memory grows. For this experiment, we populated the aforementioned `Spew` workloads with growing address spaces by leveraging the Spew [16] tool; that is, given an address space ranging from 16 GB to 1 TB, 50 K requests, whose size was a multiple of 8 KB and determined between 8 KB and 1 MB, formed each of the synthetic workloads.

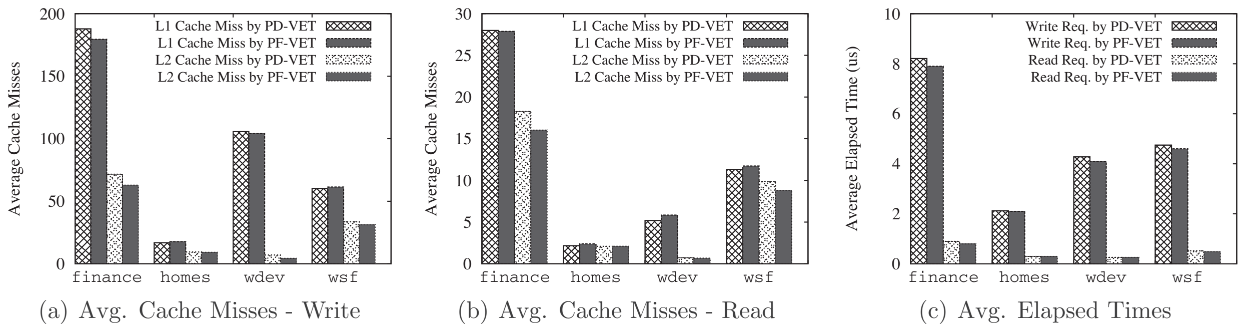


Fig. 13. Performance comparison between PD-VET and PF-VET.

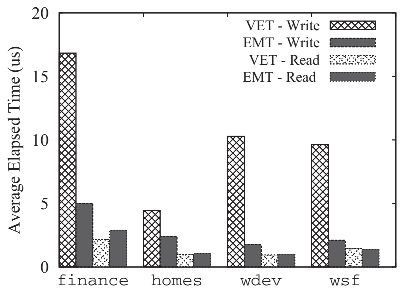


Fig. 14. Mapping time comparison.

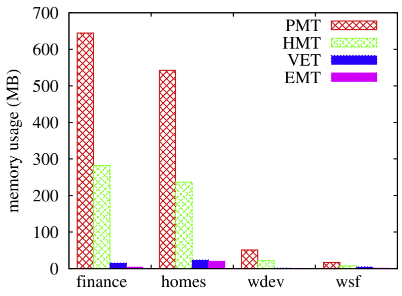


Fig. 15. Memory overhead comparison.

Fig. 16 exhibits the memory consumed by all the schemes. Because of a huge difference observed, the usage is plotted in log scale. As the space got larger, the conventional schemes suffered from enormous memory overhead, compared with our VET/EMT schemes that rather remained flat. In particular, the 1 TB address space widened the difference between the PMT and our extent schemes by even up to two orders of magnitude. (Again, EMT achieves better space reduction by approximately a factor of 7, compared to VET).

In short, the greater the capacity of the devices, the more memory the traditional techniques need proportionally, but the memory used by our extent mapping schemes is not dependent on the increasing space.

7. Related work

Waldvogel et al. [22] propose an algorithm of obtaining the best matching IP prefix using a binary search on a routing table organized by prefix lengths. In fact, the prefix and the routing table correspond to the concepts of a canonical extent and a virtual trie discussed in the article. Thus, his work presented in network domain sounds similar to VET.

However, the main differences between VET and Waldvogel's work can be summarized in the following. First, extents stored in a virtual trie must remain non-overlapping (disjoint) at all times. This is obvious, since in flash memory devices the most recent mapping information must be kept, invalidating the out-of-date one in an overlapping region if any. In his work, on the other hand, prefixes having the common prefix are allowed to co-exist in the routing table. Secondly, his work needs to use *backtracking* to find the best, correct matching prefix unless the longest one is found. The backtracking is not needed in the VET scheme, as the search for a given start address gets terminated immediately unless any extent containing the address is found. (Again, a new search can be triggered if any more addresses are to be examined, but this does not mean the backtracking.) Thirdly, an arbitrary prefix itself retains *canonical* property, while a given extent does not have to be canonical. Recall that in the VET scheme any non-canonical extent can be given, and it gets broken into several canonical extents at the time of insertion. Lastly, the routing table in his work is assumed to be *static*; namely, it does not change much over time. However, extents in a virtual trie can be updated dynamically.

A tree data structure may also be taken into account for the extent mapping like EMT. Segment tree [10] is well-known as a tree data structure for storing line segments (or intervals). However, it is fundamentally a *static* structure where the intervals already stored cannot be modified; thus, it is not appropriate to implement the extent mapping table where existing extents can be updated dynamically if a given one overlaps them.

Interval tree [23] is also an ordered tree data structure designed to store intervals. It can be built by leveraging a binary search tree. Hence, it might be considered for the extent-based mapping.

However, EMT is fundamentally different from the interval tree [23] in two-folds. First, EMT disallows any extent to overlap each other at the state of storage. Meanwhile, overlapping intervals (analogously similar to *extent*) can co-exist in the interval tree. Next, the interval tree stores intervals at leaf nodes, whereas any input extent can be stored at any node in EMT.

While page mapping [2] exposes the least mapping overhead ($O(1)$), the VET/EMT schemes can minimize the overhead just within $O(\log\log|U|)$ and $O(n\log n)$. As far as memory reduction is concerned, our extent mapping schemes outperform page mapping [2] by up to an order of magnitude in our experiments using real workloads, as demonstrated in the evaluation. Furthermore, extent mapping scheme is very favorable to the following case. Suppose that a 2 GB movie clip file gets sequentially written over a whole 128 GB SSD using page mapping [2]. Somehow, its page mapping table should be large (or 128 MB) enough to cover the whole address space, independent of the actual use of the addresses. In contrast, a virtual trie can simply keep the mapping information for the file using only tens of bytes.

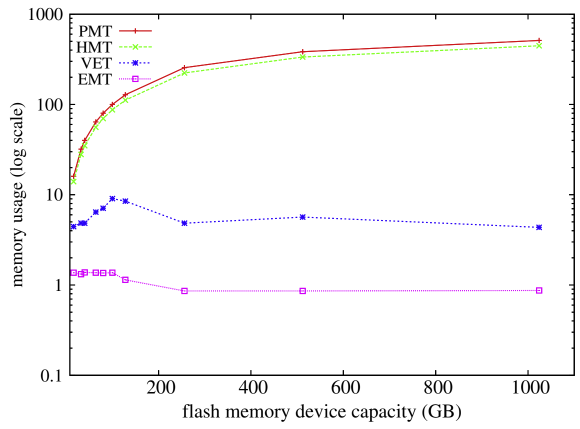


Fig. 16. Memory usage over growing address space.

The tree-based structure [24] sounds similar to our EMT scheme. However, given k 1-page writes, their scheme ends up storing a total of 2^k nodes in their tree structure as the actual mapping data are stored only at *leaf* node level, which is very space-inefficient. On the contrary, our EMT scheme needs only k nodes, as any node in EMT can store extent mapping data.

Demand-based page-level or block-level address mapping algorithms [24–27] are also proposed to reduce memory footprint by loading mapping entries into RAM on demand. However, the performance of the demand-based approaches is mainly dependent on temporal locality of workloads. In contrast, our extent mapping schemes are not dependent on such temporal locality in workloads. Also, using the demand-based approach is an orthogonal issue in our schemes. That is because the proposed VET/EMT can benefit from the demand-based approach as well when DRAM is not enough to store all the mapping information. This works by caching only the needed extent entries in memory while keeping the original mapping information on flash.

Pure block mapping [3] typically needs the least amount of memory. However, it cannot enjoy flexible utilization of physical flash pages, as the logical page number offset must be fixed within a block. Hence, garbage collection overhead is extremely high. Block mapping [3], thus, is not purely used in practice.

Hybrid mapping [4–7] to overcome such drawbacks has been also proposed. Still, it is not so flexible as page mapping. It even fails to gain more memory reduction than our extent mapping schemes, as shown in the experiments. Also, if free log pages are not left any more, the hybrid scheme needs to select a victim log block. Accordingly, the scheme needs to merge into relevant data blocks valid data from those pages in the victim block. Thus, this merge operation is very *expensive*. Again, extent mapping inherits flexibility of the page mapping scheme with less memory but minor mapping overhead. Our schemes support an memory-efficient, scalable address mapping for the extent mapping.

8. Conclusions

Flash memory devices have been remarkably advancing in the past few years. However, as their address space grows, the conventional mapping schemes, – page or hybrid ones – suffer from substantial memory overhead. To overcome this limitation, we presented novel extent mapping schemes – VET and EMT – for flash memory devices.

Our schemes treat all I/O requests as extents and dynamically manages them. In spite of the negligible mapping overhead, we observed in our experiments that the VET and EMT schemes could gain substantial memory reduction by up to an order of magnitude in comparison with the traditional mapping schemes. In addition, both schemes revealed excellent scalability over growing address spaces. These promising results demonstrate that our extent mapping schemes will accelerate the emergence of flash memory devices with much less memory but little performance degradation.

Acknowledgment

This work is supported by the National Science Foundation (NSF Grant No. IIS-0848503).

References

[1] Intel Corporation, Understanding the Flash Translation Layer (FTL) Specification, App. Note AP-684, 1998.
[2] A. Birrell et al., A design for high-performance flash disks, ACM SIGOPS Oper. Syst. Rev. 41 (2007) 88–93.
[3] A. Ban, Flash File System, 1995. US Patent 5404485.
[4] A. Ban, Flash File System Optimized for Page-Mode Flash Technologies, 1999. US Patent 5937425.
[5] S.-W. Lee et al., A log buffer-based flash translation layer using fully-associative sector translation, ACM Trans. Embed. Comput. Syst. 6 (2007) 1–27.
[6] D. Jung et al., Superblock FTL: a superblock-based flash translation layer with a hybrid address translation scheme, ACM Trans. Embed. Comput. Syst. 9 (2010) 1–41.
[7] S. Lee et al., LAST: Locality-Aware Sector Translation for NAND flash memory-based storage systems, ACM SIGOPS Oper. Syst. Rev. 42 (2008) 36–42.
[8] Y. Lee et al., μ-FTL: a memory-efficient flash translation layer supporting multiple mapping granularities, in: Proceedings of the 8th ACM & IEEE International Conference on Embedded Software (EMSOFT'08), ACM, pp. 21–30.
[9] N. Agrawal et al., Design tradeoffs for SSD performance, in: Proceeding of the USENIX ATC 2008, USENIX, pp. 57–70.
[10] J.L. Bentley, Solutions to Klees rectangle problems, Technical Report, Carnegie-Mellon University, Pittsburgh, PA, USA, 1977.
[11] D.E. Willard, Log-logarithmic worst-case range queries are possible in space $\theta(N)$, Inform. Process. Lett. 17 (1983) 81–84.
[12] P. van Emde Boas et al., Design and implementation of an efficient priority queue, Theory Comput. Syst. 10 (1976) 99–127.
[13] UMass Trace Repository, OLTP Application I/O Trace, 2011. http://traces.cs.umass.edu.
[14] D. Narayanan et al., Write off-loading: practical power management for enterprise storage, in: Proceeding of the 6th USENIX Conference on FAST, USENIX, pp. 253–267.
[15] Storage Networking Industry Association, Block I/O Trace Repository, 2012. http://iotta.snia.org/tracetypes/3.
[16] A. Patterson, Spew: An I/O Performance Measurement and Load Generation Tool, 2010. http://spew.berlios.de/.
[17] C. Clark, A Hash Table in C, 2010. http://www.cl.cam.ac.uk/cwc22/hashtable/.
[18] R. Pagh et al., Cuckoo hashing, Eur. Symp. Algor. (ESA) 9 (2001) 1–41.
[19] T. Hanson, UTHASH: A Hash Table for C Structures, 2012. http://uthash.sourceforge.net/.
[20] University of Tennessee, Performance Application Programming Interface, 2012. http://icl.cs.utk.edu/papi/.
[21] Valgrind User Manual, Cachegrind: A Cache and Branch-Prediction Profiler, 2012. http://valgrind.org/docs/manual/cg-manual.html.
[22] M. Waldvogel et al., Scalable high speed IP routing lookups, ACM SIGCOMM Comput. Commun. Rev. 27 (1997) 25–36.
[23] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., MIT Press and McGraw-Hill, Cambridge, MA, USA, 2001.
[24] L.-P. Chang et al., An efficient management scheme for large-scale flash-memory storage systems, in: Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004), ACM, pp. 862–868.
[25] A. Gupta et al., DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings, in: Proceeding of the 14th International Conference on ASPLOS, ACM, pp. 229–240.
[26] Zhiwei Qin et al., A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems, in: Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011), IEEE, pp. 157–166.
[27] Zhiwei Qin et al., Demand-based block-level address mapping in large-scale NAND flash storage systems, in: Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2010), IEEE/ACM/IFIP, pp. 173–182.

Young-Kyoon Suh received the B.S. degree in computer science from Kyungpook National University, Korea in 2003, and the M.S. degree in computer science from KAIST, Korea in 2005. He is currently pursuing the Ph.D. degree in computer science at the University of Arizona, USA. His research interest is in flash-based database technology, science of databases, micro-specialization, and concrete complexity.

Bongki Moon received the M.S. and B.S. degrees in computer engineering from Seoul National University, Korea, in 1985 and 1983, and the Ph.D. degree in computer science from University of Maryland, College Park in 1996. He is a professor of Computer Science & Engineering at Seoul National University. Prior to that, he had been a professor of Computer Science at the University of Arizona from 1997 till early 2013. He was also on the research staff for Samsung Electronics and Samsung Advanced Institute of Technology, Korea, from 1985 to 1990. He received an NSF CAREER Award in 1999 for his work on distributed cooperative web server design. His research interests include flash memory database systems, XML indexing and query processing, and information streaming and dissemination.

Alon Efrat received the B.Sc. degree in applied mathematics and M.Sc. degree in computer science from the Technion, Haifa, Israel, and the Ph.D. degree in computer science from the Tel-Aviv University, Tel-Aviv, Israel, in 1998. He is currently an Associate Professor with the University of Arizona, Tucson, AZ. He is a member of the Editorial Board of the International Journal of Computational Geometry & Applications and the Transactions on Algorithms Engineering. He was a Postdoctoral Research Assistant with the Stanford University and with IBM Almaden Research Center. Dr. Efrat is the recipient of the CAREER Award (2004) by National Science Foundation due to his work on ''Pattern Matching, Realistic Input Models and Sensor Placement. Useful Algorithms in Computational Geometry.''

Jin-Soo Kim received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University, Republic of Korea, in 1991, 1993, and 1999, respectively. He is currently an associate professor at Sungkyunkwan University. Before joining Sungkyunkwan University, he was an associate professor at Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of the research staff, and with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.

Sang-Won Lee received the Ph.D. degree from the Computer Science Department of Seoul National University in 1999. He is an associate professor with the School of Information and Communication Engineering at Sungkyunkwan University, Suwon, Korea. Before that, he was a research professor at Ewha Women University and a technical staff at Oracle, Korea. His research interest include flashbased database technology.