Extent Mapping Scheme for Flash Memory Devices

Young-Kyoon Suh, Bongki Moon, Alon Efrat
Dept. of Computer Science
University of Arizona
Tucson, Arizona, USA, 85721
{yksuh, bkmoon, alon}@cs.arizona.edu

Jin-Soo Kim, Sang-Won Lee
School of Info. & Comm. Engr.
Sungkyunkwan University
Suwon, 440-746, Korea
{jinsookim, swlee}@skku.edu

***Abstract*—Flash memory devices commonly rely on traditional address mapping schemes such as page mapping, block mapping or a hybrid of the two. Page mapping is more flexible than block mapping 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 a novel mapping scheme that is fundamentally different from the existing mapping strategies. We call this new scheme *Virtual Extent Trie (VET)*, as it manages 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, VET consumes much less memory to store mapping information and still remains as flexible as page mapping. We observed in our experiments that VET reduced memory consumption by up to an order of magnitude in comparison with the traditional mapping schemes for several real world workloads. The VET scheme also scaled well with increasing address spaces by synthetic workloads. With a binary search mechanism, VET limits the mapping time to $O(\log\log|U|)$, where U denotes the set of all possible logical addresses. Though the asymptotic mapping cost of VET 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.**

I. 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]. Generally, FTL 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 4KB) or a *block* (usually 256KB) 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 4TB flash memory SSD, for example, the size of its mapping table can be as large as 4GB.

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 64MB. 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 (*eg*, 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 feasible at the cost of over-provisioned flash memory (typically about 30% of usable capacity) and increased mapping latency.

In this paper, we propose a novel mapping scheme that is fundamentally different from the existing mapping strategies. We call this new scheme *Virtual Extent Trie (VET)*, as it manages 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. By storing extents instead of individual addresses in requests, VET consumes much less memory space to store the mapping information and still remains as flexible as page mapping. Also, the VET scheme works regardless of underlying flash architecture by either single or multi-channels [8], since any type of physical address information returned after flash writes can be simply stored with the extents. The preliminary results show that VET can reduce memory usage by up to an order of magnitude compared with conventional mapping schemes for real world workloads.

With a binary search mechanism, VET limits the search time for an extent to $O(\log\log|U|)$, where U denotes the set of all possible logical addresses. Though the asymptotic cost of mapping by VET is 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 rest of this paper is organized as follows. Section II outlines the design of the VET scheme. Section III proposes the algorithms of the VET scheme. Subsequently, Section IV presents the performance evaluation results using real world and synthetic workloads. Section V provides related work. Lastly, Section VI concludes this paper.

II. Design Principle

A. 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 scheme we propose in this paper 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* 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 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, VET 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, VET stores mapping information at the varying degree of granularity, which is determined solely by each individual write request. As will be presented below in detail, the mapping information of a write request is represented by one or more *canonical extents*. Since the size of a canonical extent can only be bounded by the size of an entire logical address space, VET can maintain mapping information in a more concise manner and reduce the size of a mapping table considerably.

B. Virtual Trie

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 '$\star$' 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 '$\star$' bits. A binary string with a few trailing '$\star$' bits in fact represents an extent whose length is a power of two. For example, an 8-bit binary string $\texttt{0010}\star\star\star\star$ can be used to represent an extent whose logical start address is $\texttt{00100000}$ 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. 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 $\log_2 l$ '$\star$' bits. For example, a canonical extent $\langle 8, 4\rangle$ can be represented by an 8-bit binary string (for simplicity) as below:

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

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 paper, for convenience of our discussion it is just omitted.

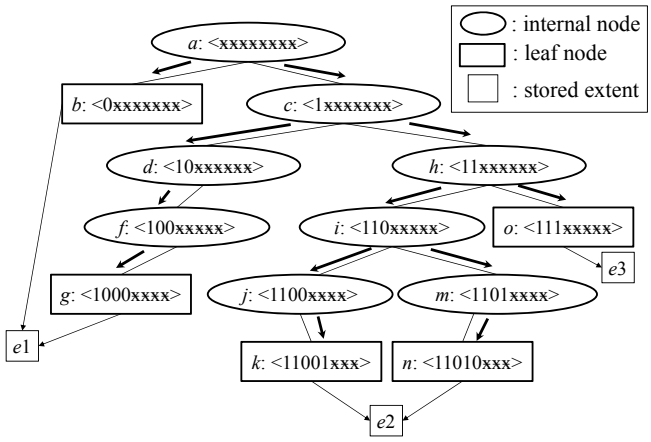


Fig. 1. Virtual Trie by Given Extents $e1$, $e2$ and $e3$

Figure 1 shows a virtual trie to store the input extents in the canonical form as follows:

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

The VET scheme finds a set of canonical extents for them, using node keys in top-down fashion, as shown in Figure 1. For instance, given $e1$, the node a's key ($\langle\star\star\star\star\star\star\star\star\rangle$) 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 the children, nodes *b* and *c*, can completely contain $e1$; thus, $e1$ is broken into smaller extents that retain candidate canonical ones. In this way, all the canonical extents of $e1$ can be located, and $e1$ is eventually kept by nodes *b* and *g*. Once the other input extents $e2$ and $e3$ are stored, the virtual trie will look like Figure 1.

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* store $e1$, leaf nodes *k* and *n* have $e2$, and leaf node *o* keeps $e3$ in Figure 1. 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 III-B.

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 Figure 1 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.

C. 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.

III. 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 II, 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.

A. 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.

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 III-B. 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.

a) LIS (Linear Insertion Scheme): 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, the VET scheme 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 Figure 1 provided in Section II-B. For $e3 = \langle 111\star\star\star\star\star\rangle$, for instance, the VET scheme using LIS (linearly) inserts internal nodes *a*, *c*, and *h* followed by leaf node *o* pointing to $e3$. 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 $e3$.

2) Deleting an Extent: If an extent generated 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.

Figure 2 exemplifies the partial invalidation. Suppose that a given extent $e4 = \langle 1100100\star\rangle$ arrives at the trie. Because $e4$ overlaps $e2$ existing in the trie, $e2$ is decomposed into the following extents:

$e4$: $\{\langle 1100100\star\rangle\}$
$e5$: $\{\langle 1100101\star\rangle, \langle 110011\star\star\rangle, \langle 11010\star\star\star\rangle\}$.

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 with e.
4      else
5          if node == internal then Delete the subtrie rooted
           from node.
6          Have node keep e, removing the node's old extent if
           any.
7      end
8  else
9      if node == NULL then Create an internal node.
10     else if node == leaf then node switches to internal.
11     Recursively find the rest of the canonical extents of e using
       the left or right child of key.
12 end
```

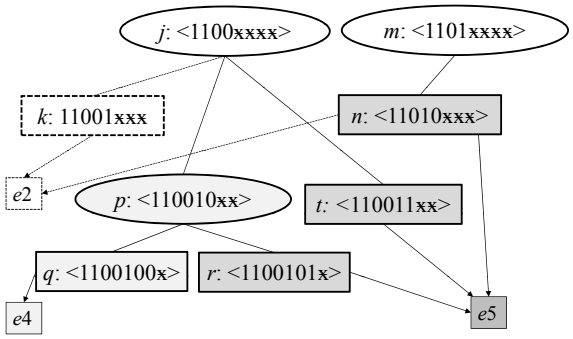


Fig. 2. Example of Partial Invalidation

By the partial invalidation, the incoming extent $e4$ overwrites one of the canonical extents of $e2$, and thus, the VET scheme creates internal node *p* and leaf node *q* for $e4$. In addition, now that the new extent $e5$ comprises the other three canonical extents, VET generates leaf nodes *r* and *t*, and simply keeps leaf node *n*, linking it to $e5$. Lastly, $e2$ is evicted with leaf node *k*, which used to retain one of the canonical extents ($\langle 11001{*}{*}{*}\rangle$) of $e2$.

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

B. 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

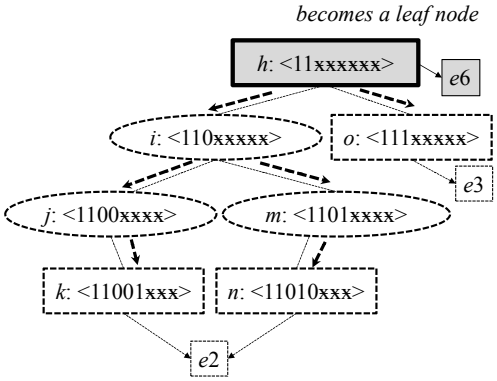


Fig. 3. Subtrie Deletion

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.

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

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 $\langle 1100{*}{*}{*}{*}\rangle$, the logical-to-physical address translation of `11001100` can be obtained immediately from the leaf node.

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

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

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$11001***$\rangle$

In a nutshell, the direction of the next search is determined by the existence of an internal node storing the current search key. Figure 4 depicts the search process along the path of the virtual trie.

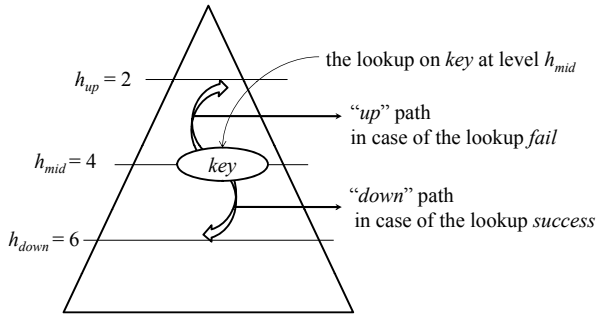


Fig. 4. Binary Search on VET Levels

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: 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
```

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

C. 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 Figure 1 has its following ancestors:

internal node a ********
internal node c 1*******
internal node h 11******
internal node i 110*****
internal node j 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 an indispensable 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).

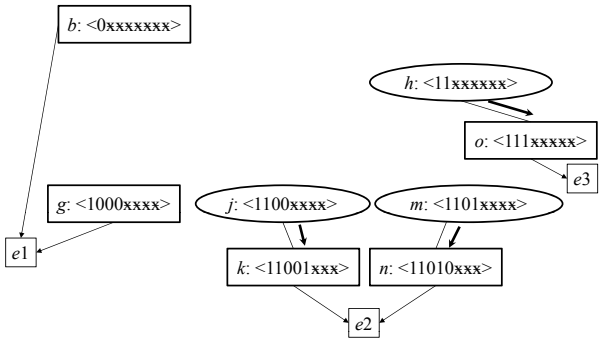


Fig. 5. Virtual Trie via BIS by Given Extents $e1$, $e2$ and $e3$

Figure 5 represents the virtual trie made by BIS. Most of the internal nodes are not seen, as opposed to Figure 1. 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 outdated 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  else Do line 9-11 in Algorithm 1.
```

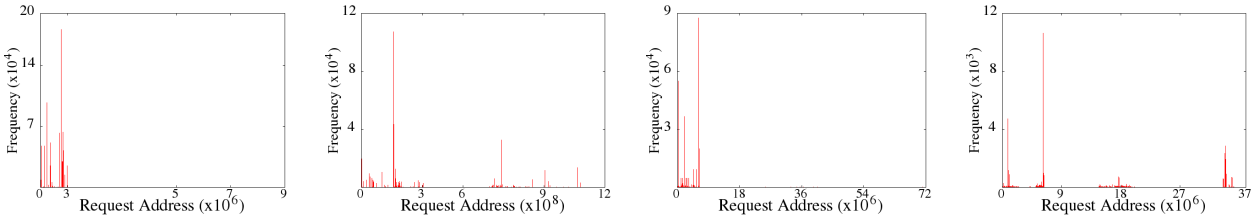
Algorithm 3 shortly describes the process of BIS. Most of the parts are similar to those of Algorithm 1. The BIS algorithm, however, is slightly different, since it should locate and remove a leaf node with an invalid extent that may lie above a newly inserted internal node, as illustrated at line 2. For instance, in Figure 2 node k with $e2$ 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.

Type	Name	Desc.	Addr. Space	# of Writes	# of Reads
				Avg. Write Size	Avg. Read Size
Real	finance	OLTP application[9]	644GB	8.2M	2.5M
				3.7KB	2.5KB
	homes	MS Exchange Servers[10], [11]	542GB	16M	0.4M
				4KB	12.0KB
	wdev	MS Exchange Servers[10], [11]	51GB	1.1M	0.23M (12.6KB)
				8.2KB	12.6KB
	wsf	Web surfing activity on PC	32GB	0.3M	93K
				22.5KB	19.8KB
Synthetic	Spew	Spew[12] workload generator	16GB∼1TB	50K	0
				83.8KB∼376.8KB	N/A

TABLE I
REAL/SYNTHETIC WORKLOADS USED FOR THE PERFORMANCE EVALUATION

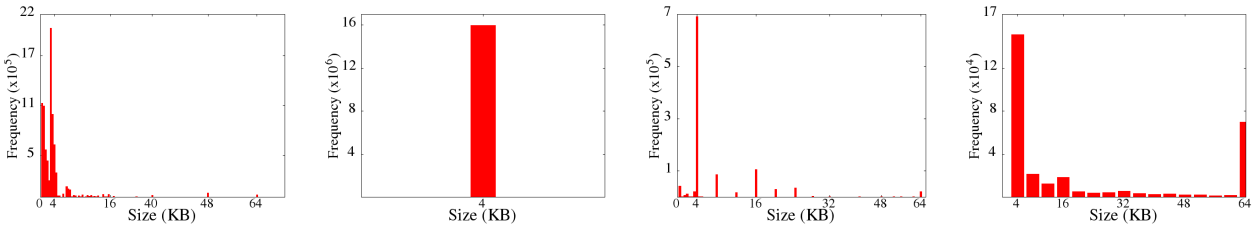
	Type	finance		homes		wdev		wsf	
Memory Usage (MB)	BIS	15.07		23.25		1.15		4.69	
	LIS	16.89		28.50		1.63		6.03	
		Write	Read	Write	Read	Write	Read	Write	Read
Avg. Elapsed Time (us)	BIS	16.84	2.17	4.43	0.99	10.29	0.95	9.64	1.44
	LIS	24.63	2.47	8.25	1.07	9.44	1.00	12.97	1.53

TABLE II
MEMORY FOOTPRINT AND AVERAGE ELAPSED TIMES



(a) finance (b) homes (c) wdev (d) wsf

Fig. 6. Histograms on Write Request Address



(a) finance (b) homes (c) wdev (d) wsf

Fig. 7. Histograms on Write Request Size

IV. EXPERIMENTS

In this section, we describe our environmental settings, and present our simulation results.

A. Environment Settings

We implemented all of the proposed algorithms in C language, and conducted all experiments on a low-end 32-bit machine with a Pentium 4 CPU 3.00GHz processor and 2GB memory, running the Linux system with a 2.6.32-32-generic kernel version.

The description of the workloads used for the performance evaluation is given in Table I. The real world traces - finance, homes, wdev - were obtained from public I/O trace repositories [9], [11], and wsf was the real workload extracted from the web activity. The Spew trace gained by a workload generator [12] comprises only write requests. As in Section IV-B3, this trace was specially used for the analysis of memory overhead with an increasing address space. We also provide the histograms about write request address and size in the real workloads, as illustrated in Figures 6 and 7.

B. Performance Evaluation

This section presents our performance evaluation results.

1) Overall Analysis: We analyze the overall performance of the VET scheme on the real world traces, with respect to memory usage and canonical extents.

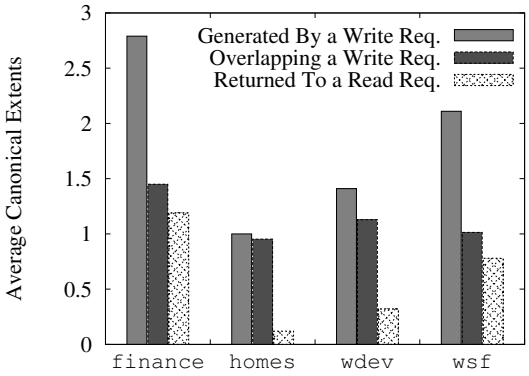


Fig. 8. Canonical Extent Statistics on a Given I/O Request

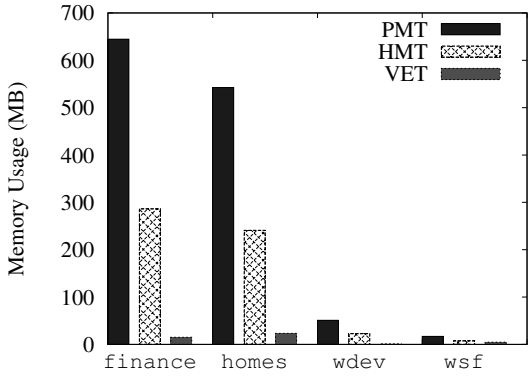


Fig. 9. Memory Overhead Comparison

a) Memory Consumption: As seen in Table II, the amount of memory needed to treat each trace was relatively very small, considering the address space of the trace. (Certainly, there was some mapping overhead.) Specifically, the VET scheme only used at most about 17MB memory for processing the `finance` trace, which had the largest address space, irrespective of whichever insertion methods (LIS or BIS) was employed. Apparently, `homes` consumed the greatest deal of memory among them. It was because the `homes` workload revealed the most widespread range of request addresses, as illustrated in Figure 6, and it had far more write requests than any other workload, as indicated in Table I. This implies that a large deal 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 tended to be accessed repeatedly, 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 Figure 6(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`.

b) Canonical Extents: Figure 8 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, respectively. The third one indicates average canonical extents responded to an inquiry extent.

As shown, the `finance` and `homes` workloads required the largest and fewest average numbers of canonical extents involving an I/O request, correspondingly. In the other traces, much fewer average canonical extents per read request were answered for `wdev` than for `wsf`. In the case of writes, meanwhile, the `wdev` trace revealed slightly more average extents overlapping but about half of ones generated per write request than the `wsf` trace.

The observations above are fairly consistent with the average elapsed time results shown in Table II. The fewer canonical extents involved, the faster the average elapsed times on a given I/O request.

Note that all the traces on average generated at most three canonical extents for an input extent, as shown in Figure 8. This demonstrates that the theoretic bound ($2\log|U|$) on the average number of canonical extents is overly pessimistic in practice. The proof will be provided in the extended version of the paper.

2) VET vs. Page/Hybrid Mapping Tables: As shown in Figure 9, the VET scheme used much less memory than a page mapping table (PMT) [2] or a hybrid mapping table (HMT) [4], [5], [6], [7]. Specifically, when treating the workloads, it consumed only about 2.3%, 4.3%, 2.3%, and 27% of the total memory required by the PMT in order. Even compared with the HMT, the VET scheme used only about 5.3%, 9.8%, 5.2% and 64% of the HMT's total memory in order. This is attributed to the fact that the memory required by VET 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 their addresses are.

For instance, the average write request size of `finance` or `homes` was not so small that at best two internal nodes were created. Furthermore, both of the workloads had huge address spaces more than 0.5TB, but their writes touched only a few portions of each of the spaces, which did not create much mapping information in VET. Thus, a large amount of the memory could be saved.

The `wdev` trace had a small address space, but again many of the writes, whose size was also optimized to VET, tended to use the same addresses accessed before, thereby inducing no additional memory consumption. In the case of `wsf`, it had the smallest address space, and even fewer portions of the space

were left intact unlike the other traces. Hence, the memory reduction on `wsf` was not so great as that of the others.

3) Scalability Test: For this experiment, we populated the `Spew` workloads by leveraging the Spew [12] tool; that is, given an address space increasing from 16GB to 1TB, 50K requests, whose size was a multiple of 8KB and determined between 8KB and 1MB, formed each of the workloads.

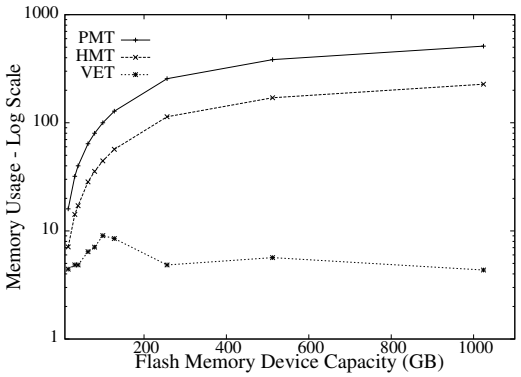


Fig. 10. Memory Usage over Growing Address Space

Figure 10 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 the VET scheme that rather remained flat. In particular, the 1TB address space widened the gap between the PMT and VET by up to two orders of magnitude.

V. Related Work

Waldvogel *et al.* [13] propose an algorithm of obtaining the best matching IP prefix through 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 mentioned in the paper. His work presented in network domain, therefore, sounds similar to the VET scheme.

However, the main differences between VET and Waldvogel's work can be summarized as follows. First, extents stored in a virtual trie must remain non-overlapping (disjoint) at all times, so that the most recent mapping information should be kept by getting the out-of-date one invalidated. In his work, on the contrary, prefixes having the common one can 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. In the VET scheme, the backtracking does not take place, as the search for a given logical address is terminated immediately if any extent containing the address cannot be located. Thirdly, an arbitrary prefix itself retains *canonical* property, while a given extent does not have to be canonical. Lastly, the routing table in his work is assumed to be *static*; namely, it does not change much over time. Meanwhile, VET is capable of handling the *dynamic* update of extents.

Page mapping [2] exhibits the least mapping overhead, $O(1)$. The VET scheme also defends the search time just within $O(\log\log|U|)$, but it still falls behind page mapping [2] with respect to the mapping time.

DFTL [14] was proposed to attack the random write problem in flash memory devices by leveraging temporal locality. However, it still sticks to page mapping, so it cannot escape from the burden of the memory overhead along with an increasing address space.

Pure block mapping [3] typically earns the most space reduction. It, however, cannot enjoy flexible utilization of physical flash pages, because of the problem of the logical page number offset that must be fixed within a block. As a result, garbage collection overhead is extremely high, and it is not purely used in practice.

Hybrid mapping [4], [5], [6], [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 VET, as shown in our experiments.

VI. Conclusion

In this paper, we proposed *VET*, a novel extent mapping scheme for flash memory devices. Compared with the traditional mapping schemes, VET gained substantial space reduction by even up to an order of magnitude on real traces, and it also scaled very well with increasing spaces by synthetic workloads. Such considerable memory reduction was achieved with negligible mapping overhead.

VII. Acknowledgments

This work was supported by the National Science Foundation under NSF Grant numbers IIS-0848503.

References

[1] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification," App. Note AP-684, 1998.
[2] Birrell, A. et al., "A Design for High-Performance Flash Disks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 88–93, Apr. 2007.
[3] Ban, A., "Flash File System," April 1995, uS Patent 5404485.
[4] ——, "Flash File System Optimized for Page-Mode Flash Technologies," August 1999, uS Patent 5937425.
[5] Lee, S-W. et al., "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation," *ACM Trans. on Embedded Computing Systems*, vol. 6, no. 3, pp. 1–27, 2007.
[6] Jung, D. et al., "Superblock FTL: A Superblock-Based Flash Translation Layer with a Hybrid Address Translation Scheme," *ACM Trans. on Embedded Computing Systems*, vol. 9, no. 4, pp. 1–41, 2010.
[7] Lee, S. et al., "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems," *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 36–42, 2008.
[8] Agrawal, N. et al., "Design Tradeoffs for SSD Performance," in *Proceeding of the USENIX ATC 2008*. USENIX, 2008, pp. 57–70.
[9] UMass Trace Repository, "OLTP Application I/O Trace," 2011, http://traces.cs.umass.edu.
[10] Narayanan, D. et al., "Write Off-loading: Practical Power Management for Enterprise Storage," in *Proceeding of the 6th USENIX Conf. on FAST*. USENIX, 2008, pp. 253–267.
[11] Storage Networking Industry Association, "Block I/O Trace Repository," 2011, http://iotta.snia.org/tracetypes/3.
[12] Patterson, A., "Spew: An I/O Performance Measurement and Load Generation Tool," 2011, http://spew.berlios.de/.
[13] Waldvogel, M. et al., "Scalable High Speed IP Routing Lookups," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 4, pp. 25–36, 1997.
[14] Gupta, A. et al., "DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings," in *Proceeding of the 14th Intl' Conf. on ASPLOS*. ACM, 2009, pp. 229–240.