De-indirection for Flash-based SSDs with Nameless Writes

Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison

Abstract

We present *Nameless Writes*, a new device interface that removes the need for indirection in modern solid-state storage devices (SSDs). Nameless writes allow the device to choose the location of a write; only then is the client informed of the *name* (i.e., address) where the block now resides. Doing so allows the device to control block-allocation decisions, thus enabling it to execute critical tasks such as garbage collection and wear leveling, while removing the need for large and costly indirection tables. We demonstrate the effectiveness of nameless writes by porting the Linux ext3 file system to use an emulated nameless-writing device and show that doing so both reduces space and time overheads, thus making for simpler, less costly, and higher-performance SSD-based storage.

1 Introduction

Indirection is a core technique in computer systems [28]. Whether in the mapping of file names to blocks, or a virtual address space to an underlying physical one, system designers have applied indirection to improve system performance, reliability, and capacity for many years.

For example, modern hard disk drives use a modest amount of indirection to improve reliability by hiding underlying write failures. When a write to a particular physical block fails, a hard disk will *remap* the block to another location on the drive and record the mapping such that future reads will receive the correct data. In this manner, a drive transparently improves reliability without requiring any changes to the client above.

Indirection is particularly important in the new class of flash-based storage commonly referred to as Solid State Devices (SSDs). In modern SSDs, an indirection map in the Flash Translation Layer (FTL) enables the device to map writes in its virtual address space to any underlying physical location [11, 14, 16, 19, 21, 22].

FTLs use indirection for two reasons: first, to transform the erase/program cycle mandated by flash into the more typical write-based interface via copy-on-write techniques, and second, to implement *wear leveling* [18, 20], which is critical to increasing SSD lifetime. Because a flash block becomes unusable after a certain number of erase-program cycles (10,000 or 100,000 cycles according to manufacturers [8, 15]), such indirection is needed to spread the write load across flash blocks evenly and thus ensure that no particularly popular block causes the device to fail prematurely.

Unfortunately,the indirection such as found in many FTLs comes at a high price, which manifests as performance costs, space overheads, or both. If the FTL can flexibly map each virtual *page* in its address space (assuming a typical page size of 2 KB), an incredibly large indirection table is required. For example, a 1-TB SSD would need 2 GB of table space simply to keep one 32-bit pointer per 2-KB page of the device. Clearly, a completely flexible mapping is too costly; putting vast quantities of memory (usually SRAM) into an SSD is prohibitive.

Because of this high cost, most SSDs do not offer a fully flexible per-page mapping. A simple approach provides only a pointer per *block* of the SSD (a block typically contains 64 or 128 2-KB pages), which reduces overheads by the ratio of block size to page size. The 1-TB drive would now only need 32 MB of table space, which is more reasonable. However, as clearly articulated by Gupta et al. [16], block-level mappings have high performance costs due to excessive garbage collection.

As a result, the majority of FTLs today are built using a hybrid approach, mapping most data at block level and keeping a small page-mapped area for updates [11, 21, 22]. Hybrid approaches keep space overheads low while avoiding the high overheads of garbage collection, at the cost of additional device complexity. Unfortunately, garbage collection can still be costly, reducing the performance of the SSD, sometimes quite noticeably [16]. Regardless of the approach, FTL indirection incurs a significant cost; as SSDs scale, even hybrid schemes mostly based on block pointers will become infeasible.

In this paper, we introduce nameless writes, an approach that removes most of the costs of indirection in flash-based SSDs while still retaining its benefits. Our approach is a specific instance of *de-indirection*, in which an extra layer of indirection is removed. Unlike most writes, which specify both the *data* to write as well as a *name* (usually in the form of a logical address), a nameless write simply passes the data to the device. The device is free to choose any underlying physical block for the data; after the device *names* the block (i.e., decides where to write it), it informs the client of its choice. The client then can record the name for future reads.

One potential problem with nameless writes is the recursive update problem: if all writes are nameless, then any update to the file system requires a recursive set of updates up the file-system tree. To circumvent this problem, we introduce a *segmented address space*, which consists of a (large) physical address space for nameless writes, and a (small) virtual address space for traditional named writes. A file system running atop a nameless SSD can keep pointer-based structures in the virtual space; updates to those structures do not necessitate further updates up the tree, thus breaking the recursion.

Nameless writes offer great advantage over traditional writes, as they largely remove the need for indirection. Instead of pretending that the device can receive writes in any frequency to any block, a device that supports nameless writes is free to assign any physical page to a write when it is written; by returning the true name (i.e., the physical address) of the page to the client above (e.g., the file system), indirection is largely avoided, reducing the monetary cost of the SSD, improving its performance, and simplifying its internal structure.

Nameless writes (largely) remove the costs of indirection without giving away the primary responsibility an SSD manufacturer maintains: wear leveling. If an SSD simply exports the physical address space to clients, a simplistic file system or workload could cause the device to fail rather rapidly, simply by over-writing the same block repeatedly (whether by design or simply through a file-system bug). With nameless writes, no such failure mode exists. Because the device retains control of naming, it retains control of block placement, and thus can properly implement wear leveling to ensure a lengthy device lifetime. We believe that any solution that does not have this property is not viable, as no manufacturer would like to be so easily exposed to failure.

We demonstrate the benefits of nameless writes by porting the Linux ext3 file system to use a nameless SSD. Through extensive analysis on an emulated nameless SSD and comparison with different FTLs, we show the benefits of the new interface, in both reducing the space costs of indirection and improving random-write performance. Overall, we find that a nameless SSD uses a much smaller fraction of memory for indirection than a hybrid SSD while improving performance by an order of magnitude for some workloads.

The rest of this paper is organized as follows. In Section 2, we discuss the costs and benefits of indirection, and in Section 3 we present the nameless write interface. In Section 4, we show how to build a nameless-writing device. In Section 5, we describe how to port the Linux ext3 file system to use the nameless-writing interface, and in Section 6, we evaluate nameless writes through experimentation atop an emulated nameless-writing device. We discuss several related works in Section 7. Finally, in Section 8, we conclude and discuss our future work.

2 Indirection

It is said that “all problems in computer science can be solved by another level of indirection,” a quote that is often attributed to Butler Lampson. Lampson, however, gives credit for this wisdom to David Wheeler, who not only uttered these famous words, but also usually added “...but that usually will create another problem [28].”

Indirection is a fundamental technique in computer systems. Before delving into the details of nameless writes, we first present a discussion of some of the general problems and solutions in systems that use indirection. First, we discuss why many systems utilize multiple levels of indirection, a problem we term *excess indirection*. We then describe the general solution to said problem, *de-indirection*, which removes an extra layer of indirection to improve performance or reduce space overheads.

2.1 Excess Indirection

Excess indirection exists in many systems that are widely used today, as well as in research prototypes. We now discuss four prominent examples: OS virtual memory running atop a hypervisor, a file system running atop a single disk, a file system atop a RAID array, and the focus of our work, file systems atop flash-based SSDs.

An excellent example of excess indirection arises in memory management of operating systems running atop hypervisors [9]. The OS manages virtual-to-physical mappings for each process that is running; the hypervisor, in turn, manages physical-to-machine mappings for each OS. In this manner, the hypervisor has full control over the memory of the system, whereas the OS above remains unchanged, blissfully unaware that it is not managing a real physical memory. Excess indirection leads to both space and time overheads in virtualized systems. The space overhead comes from maintaining OS physical addresses to machine addresses mapping for each page and from possible additional space overhead [1]. Time overheads exist as well in cases like the MIPS TLB-miss lookup in Disco [9].

Indirection also exists in modern disks. For example, modern disks maintain a small amount of extra indirection that maps bad sectors to nearby locations, in order to improve reliability in the face of write failures. Other examples include ideas for “smart” disks that remap writes in order to improve performance (for example, by writing to the nearest free location), which have been explored in previous research such as Loge [13] and “intelligent” disks [30]. These smart disks require large indirection tables inside the drive to map the logical address of the write to its current physical location. This requirement introduces new reliability challenges, including how to keep the indirection table persistent. Finally, fragmentation of randomly-updated files is also an issue.

File systems running atop modern RAID storage arrays provide another excellent example of excess indirection. Modern RAIDs often require indirection tables for fully-flexible control over the on-disk locations of blocks. In AutoRAID, a level of indirection allows the system to keep active blocks in mirrored storage for performance reasons, and move inactive blocks to RAID to increase effective capacity [32] and overcome the RAID small-update problem [26]. When a file system runs atop a RAID, excess indirection exists because the file system maps logical offsets to logical block addresses. The RAID, in turn, maps logical block addresses to physical (disk, offset) pairs. Such systems add memory space overhead to maintain these tables and meet the challenges of persisting the tables across power loss.

The focus of our work is flash-based SSDs, and thus it is no surprise that these too exhibit excess indirection. The extra level of indirection is provided via the Flash Translation Layer (FTL). The FTL is needed for two primary reasons. First, it is used to transform reads and writes issued by the client into reads and erase/program cycles supported by actual flash chips. In particular, because of the high cost of block erases (required before programming a page within the block), FTLs map current write activity to a small set of active blocks in a log-structured fashion, thus amortizing the cost of erases. Second, the FTL enables the SSD to implement wear leveling. Repeatedly erasing and programming a particular block will render it unreadable; thus, SSDs use the indirection provided by the FTL to spread write load across blocks and thus ensure that the device has a longer lifetime.

2.2 De-indirection

Because of these costs, system designers have long sought methods and techniques to reduce the costs of excess indirection in various systems. We label the removal of excess indirection *de-indirection*.

The basic idea is simple. Let us imagine a system with two levels of mapping, and thus excess indirection. The first indirection F maps items in the A space to items in the B space: $F(A_i) \rightarrow B_j$. The second indirection G maps items in the B space to those in the C space: $G(B_j) \rightarrow C_k$. To look up item i, one performs the following "excessive" indirection: $G(F(i))$.

De-indirection removes the second level of indirection by evaluating the second mapping $G()$ for all values mapped by $F()$: $\forall\, i : F(i) \leftarrow G(F(i))$. Thus, the top-level mapping simply extracts the needed values from the lower level indirection and installs them directly.

De-indirection has been successfully applied in a few domains, most notably within hypervisors. The Turtles project [7] provides an excellent example: in a recursively-virtualized environment (with hypervisors running on hypervisors), the Turtles system installs what the authors refer to as *multi-dimensional page tables*. Their approach essentially collapses multiple page tables into a single extra level of indirection, and thus reduces space and time overheads, making the costs of recursive virtualization more palatable.

2.3 Summary

Excess indirection is common across virtual memory and storage systems. In some cases, such as with hypervisor-based memory virtualization, it is required for functionality; each OS believes it owns the same physical memory, and thus cannot share it without the indirection provided by the hypervisor. In other cases, it improves performance, as we observed with disk systems and SSDs. Another reason for indirection is modularity and code simplicity. Finally, reliability is often the reason for excess indirection, notably within a single disk to handle write failures and within an SSD to perform wear leveling.

In all cases, at least part of the reason for excess indirection is the need to keep a fixed interface between higher and lower layers of the system. Without such a constraint, one could often remove the excess indirection and thus improve the system. For example, if an OS running on a para-virtualized system [31] is modified to request a machine page from the hypervisor and then install the correct virtual-to-machine page translation in its page tables, the hypervisor is relieved of having to manage this extra level of indirection, thus improving performance and reducing space overheads.

3 Nameless Writes

In this section, we discuss a new device interface that enables flash-based SSDs to remove a great deal of their infrastructure for indirection. We call a device that supports this interface a *Nameless-writing Device*. Table 1 summarizes the nameless-writing device interface.

The key feature of a nameless-writing device is its ability to perform nameless writes; however, to facilitate clients (such as file systems) to use a nameless-writing device, a number of other features are useful as well. In particular, the nameless-writing device should provide support for a segmented address space, migration callbacks, and associated metadata. We discuss these features in this section and how a prototypical file system could use them.

3.1 Nameless Write Interfaces

We first present the basic device interfaces of *Nameless Writes*: nameless (new) write, nameless overwrite, physical read, and free.

The nameless write interface completely replaces the existing write operation. A nameless write differs from a traditional write in two important ways. First, a nameless write does not specify a target address (i.e., a name); this allows the device to select the physical location without control from the client above. Second, after the device writes the data, it returns a *physical* address (i.e., a name)

Virtual Read		
	down:	virtual address, length
	up:	status, data
Virtual Write		
	down:	virtual address, data, length
	up:	status
Nameless Write		
	down:	data, length, metadata
	up:	status, resulting physical address(es)
Nameless Overwrite		
	down:	old physical address(es), data, length, metadata
	up:	status, resulting physical address(es)
Physical Read		
	down:	physical address, length, metadata
	up:	status, data
Free		
	down:	virtual/physical addr, length, metadata, flag
	up:	status
Migration [Callback]		
	up:	old physical addr, new physical addr, metadata
	down:	old physical addr, new physical addr, metadata

Table 1: **The Nameless-Writing Device Interfaces** *The table presents the nameless-writing device interfaces.*

and status to the client, which then keeps the name in its own structure for future reads.

The nameless overwrites interface is similar to the nameless (new) write interface, except that it also passes the old physical address(es) to the device. The device frees the data at the old physical address(es) and then performs a nameless write.

Read operations are mostly unchanged; as usual, they take as input the physical address to be read and return the data at that address and a status indicator. A slight change of the read interface is the addition of metadata in the input, for reasons that will be described in Section 3.4.

Because a nameless write is an allocating operation, a nameless-writing device needs to also be informed of deallocation as well. Most SSDs refer to this interface as the *free* or *trim* command. Once a block has been freed (trimmed), the device is free to re-use it.

Finally, we consider how the nameless write interface could be utilized by a typical file-system client such as Linux ext3. For illustration, we examine the operations to append a new block to an existing file. First, the file system issues a nameless write of the newly-appended data block to a nameless-writing device. When the nameless write completes, the file system is informed of its address and can update the corresponding in-memory inode for this file so that it refers to the physical address of this block. Since the inode has been changed, the file system will eventually flush it to the disk as well; the inode must be written to the device with another nameless write. Again, the file system waits for the inode to be written and then updates any structures containing a reference to the inode. If nameless writes are the only interface available for writing to the storage device, then this recursion will continue until a root structure is reached. For file systems that do not perform this chain of updates or enforce such ordering, such as Linux ext2, additional ordering and writes are needed. This problem of recursive update has been solved in other systems by adding a level of indirection (e.g., the inode map in LFS [27]).

3.2 Segmented Address Space

To solve the recursive update problem without requiring substantial changes to the existing file system, we introduce a segmented address space with two segments (see Figure 1): the *virtual address space*, which uses virtual read, write and free interfaces, and the *physical address space*, which uses nameless read, write, overwrite, and free interfaces.

The virtual segment presents an address space from blocks 0 through $V - 1$, and is a virtual block space of size V blocks. The device virtualizes this address space, and thus keeps a (small) indirection table to map accesses to the virtual space to the correct underlying physical locations. Reads and writes to the virtual space are identical to reads and writes on typical devices. The client sends an address and a length (and, if a write, data) down to the device; the device replies with a status message (success or failure), and if a successful read, the requested data.

The nameless segment presents an address space from blocks 0 through $P - 1$, and is a physical block space of size P blocks. The bulk of the blocks in the device are found in this physical space, which allows typical named reads; however, all writes to physical space are nameless, thus preventing the client from directly writing to physical locations of its choice.

We use a virtual/physical flag to indicate the segment a block is in and the proper interface it should go through. The size of the two segments are not fixed. Allocation in either segment can be performed while there is still space on the device. A device space usage counter can be maintained for this purpose.

The reason for the segmented address space is to enable file systems to largely reduce the levels of recursive updates that would occur with only nameless writes. File systems such as ext2 and ext3 can be designed such that inodes and other metadata are placed in the virtual address space. Such file systems can simply issue a write to an inode and complete the update without needing to modify directory structures that reference the inode. Thus, the segmented address space allows updates to complete without propagating throughout the directory hierarchy.

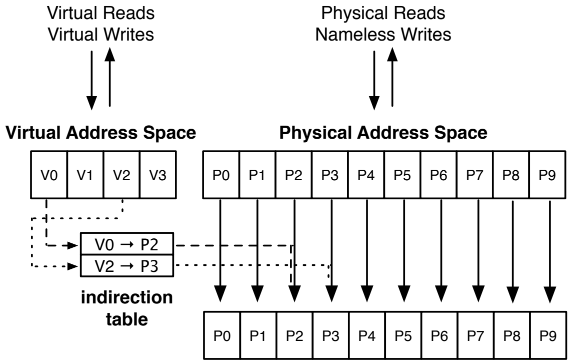

Figure 1: **The Segmented Address Space.** *A nameless-writing device provides a segmented address space to clients. The smaller virtual space allows normal reads and writes, which the device in turn maps to underlying physical locations. The larger physical space allows reads to physical addresses, but only nameless writes. In the example, only two blocks of the virtual space are currently mapped, V0 and V2, to physical blocks P2 and P3, respectively.*

3.3 Migration Callback

Several kinds of devices such as flash-based SSDs need to migrate data for reasons like wear leveling. We propose the *migration callback* interface to support such needs.

A typical flash-based SSD performs wear leveling via indirection: it simply moves the physical blocks and updates the map. With nameless writes, blocks in the physical segment cannot be moved without informing the file system. To allow the nameless-writing device to move data for wear leveling, a nameless-writing device uses *migration callbacks* to inform the file system of the physical address change of a block. The file system then updates any metadata pointing to this migrated block.

3.4 Associated Metadata

The final interface of a nameless-writing device is used to enable the client to quickly locate metadata structures that point to data blocks. The complete specification for associated metadata supports communicating metadata between the client and device. Specifically, the nameless write command is extended to include a third parameter: a small amount of metadata, which is persistently recorded adjacent to the data in a per-block header. Reads and migration callbacks are also extended to include this metadata. The associated metadata is kept with each block buffer in the page cache as well.

This metadata enables the client file system to readily identify the metadata structure(s) that points to a data block. For example, in ext3 we can locate the metadata structure that points to a data block by the inode number, the inode generation number, and the offset of the block in the inode. For file systems that already explicitly record back references, such as btrfs and NoFS [10], the back references can simply be reused for our purposes.

Such metadata structure identification can be used in several tasks. First, when searching for a data block in the page cache, we obtain the metadata information and compare it against the associated metadata of the data blocks in the page cache. Second, the migration callback process uses associated metadata to find the metadata that needs to be updated when a data block is migrated. Finally, associated metadata enables recovery in various crash scenarios, which we will discuss in detail in Section 5.7.

One last issue worth noticing is the difference between the associated metadata and address mapping tables. Unlike address mapping tables, the associated metadata is not used to locate physical data and is only used by the device during migration callbacks and crash recovery. Therefore, it can be stored adjacent to the data on the device. Only a small amount of the associated metadata is fetched into device cache for a short period of time during migration callbacks or recovery. Therefore, the space cost of associated metadata is much smaller than address mapping tables.

3.5 Implementation Issues

We now discuss various implementation issues that arise in the construction of a nameless-writing device. We focus on those issues different from a standard SSD, which are covered in detail elsewhere [16].

A number of issues revolve around the virtual segment. Most importantly, how big should such a segment be? Unfortunately, its size depends heavily on how the client uses it, as we will see when we port Linux ext3 to use nameless writes in Section 5. Our results in Section 6 show that a small virtual segment is usually sufficient.

The virtual space, by definition, requires an in-memory indirection table. Fortunately, this table is quite small, likely including simple page-level mappings for each page in the virtual segment. However, the virtual address space could be made larger than the size of the table; in this case, the device would have to swap pieces of the page table to and from the device, slowing down access to the virtual segment. Thus, while putting many data structures into the virtual space is possible, ideally the client should be miserly with the virtual segment, in order to avoid exceeding the supporting physical resources.

Another concern is the extra level of information naturally exported by exposing physical names to clients. Although the value of physical names has been extolled by others [12], a device manufacturer may feel that such information reveals too much of their "secret sauce" and thus be wary of adopting such an interface. We believe that if such a concern exists, the device could hand out modified forms of the true physical addresses, thus trying to hide the exact addresses from clients. Doing so may exact additional performance and space overheads, perhaps the cost of hiding information from clients.

4 Nameless-Writing Device

In this section, we describe our implementation of an emulated nameless-writing SSD. With nameless writes, a nameless-writing SSD can have a simpler FTL, which has the freedom to do its own allocation and wear leveling. We first discuss how we implement the nameless-writing interfaces and then propose a new garbage collection method that avoids file-system interaction. We defer the discussion of wear leveling to Section 5.6.

4.1 Nameless-Writing Interface Support

We implemented an emulated nameless-writing SSD that performs data allocation in a log-structured fashion by maintaining active blocks that are written in sequential order. When a nameless write is received, the device allocates the next free physical address, writes the data, and returns the physical address to the file system.

To support the virtual block space, the nameless-writing device maintains a mapping table between logical and physical addresses in its device cache. When the cache is full, the mapping table is swapped out to the flash storage of the SSD. As our results show in Section 6.1, the mapping table size of typical file system images is small; thus, such swapping rarely happens in practice.

The nameless-writing device handles trims in a manner similar to traditional SSDs; it invalidates the physical address sent by a trim command. During garbage collection, invalidated pages can be recycled. The device also invalidates the old physical addresses of overwrites.

A nameless-writing device needs to keep certain associated metadata for nameless writes. We choose to store the associated metadata of a data page in its Out-Of-Band (OOB) area. The associated metadata is moved together with data pages when the device performs a migration.

4.2 In-place Garbage Collection

In this section, we describe a new garbage collection method for nameless-writing devices. Traditional FTLs perform garbage collection on a flash block by reclaiming its invalid data pages and migrating its live data pages to new locations. Such garbage collection requires a nameless-writing device to inform the file system of the new physical addresses of the migrated live data; the file system then needs to update and write out its metadata. To avoid the costs of such callbacks and additional metadata writes, we propose *in-place garbage collection*, which writes the live data back to the same location instead of migrating it. A similar hole-plugging approach was proposed in earlier work [24], where live data is used to plug the holes of most utilized segments.

To perform in-place garbage collection, the FTL selects a candidate block using a certain policy. The FTL reads all live pages from the chosen block together with their associated metadata, stores them temporarily in a supercapacitor- or battery-backed cache, and then erases the block. The FTL next writes the live pages to their original addresses and tries to fill the rest of the block with writes in the waiting queue of the device. Since a flash block can only be written in one direction, when there are no waiting writes to fill the block, the FTL marks the free space in the block as unusable. We call such space *wasted space*. During in-place garbage collection, the physical addresses of live data are not changed. Thus, no file system involvement is needed.

Policy to choose candidate block: A natural question is how to choose blocks for garbage collection. A simple method is to pick blocks with the fewest live pages so that the cost of reading and writing them back is minimized. However, choosing such blocks may result in an excess of wasted space. In order to pick a good candidate block for in-place garbage collection, we aim to minimize the cost of rewriting live data and to reduce wasted space during garbage collection. We propose an algorithm that tries to maximize the benefit and minimize the cost of in-place garbage collection. We define the cost of garbage collecting a block to be the total cost of erasing the block (T_{erase}), reading (T_{page_read}) and writing (T_{page_write}) live data (N_{valid}) in the block.

$$cost = T_{erase} + (T_{page\_read} + T_{page\_write}) * N_{valid}$$

We define benefit as the number of new pages that can potentially be written in the block. Benefit includes the following items: the current number of waiting writes in the device queue (N_{wait_write}), which can be filled into empty pages immediately, the number of empty pages at the end of a block (N_{last}), which can be filled at a later time, and an estimated number of future writes based on the speed of incoming writes (S_{write}). While writing valid pages (N_{valid}) and waiting writes (N_{wait_write}), new writes will be accumulated in the device queue. We account for these new incoming writes by $T_{page\_write} * (N_{valid} + N_{wait\_write}) * S_{write}$. Since we can never write more than the amount of the recycled space (i.e., number of invalid pages, $N_{invalid}$) of a block, the benefit function uses the minimum of the number of invalid pages and the number of all potential new writes.

$$\begin{aligned} benefit = \min(&N_{invalid}, N_{wait\_write} + N_{last} \\ &+ T_{page\_write} * (N_{valid} + N_{wait\_write}) * S_{write}) \end{aligned}$$

The FTL calculates the $\frac{benefit}{cost}$ ratio of all blocks that contain invalid pages and selects the block with the maximal ratio to be the garbage collection candidate. Computationally less expensive algorithms could be used to find reasonable approximations; such an improvement is left to future work.

5 Nameless Writes on ext3

In this section we discuss our implementation of nameless writes on the Linux ext3 file system. The Linux ext3 file system is a classic journaling file system that is commonly used in many Linux distributions. It extends the Linux ext2 file system and uses the same allocation method as ext2. It provides three journaling modes: data mode, ordered mode, and journal mode. The ordered journaling mode of ext3 is a commonly used mode, which writes metadata to the journal and writes data to disk before committing metadata of the transaction. It provides ordering that can be naturally used by nameless writes, since the nameless-writing interface requires metadata to reflect physical address returned by data writes. When committing metadata in ordered mode, the physical addresses of data blocks are known to the file system because data blocks are written out first. Thus, we implemented nameless writes with ext3 ordered mode; other modes are left for future work.

5.1 Segmented Address Space

We first discuss physical and virtual address space separation and modified file-system allocation on ext3. We use the physical address space to store all data blocks and the virtual address space to store all metadata structures, including superblocks, inodes, data and inode bitmaps, indirect blocks, directory blocks, and journal blocks. We use the type of a block to determine whether it is in the virtual or the physical address space and the type of interface it goes through.

The nameless-writing file system does not perform allocation of the physical address space and only allocates metadata in the virtual address space. Therefore, we do not fetch or update group bitmaps for nameless block allocation. For these data blocks, the only bookkeeping task that the file system needs to perform is tracking overall device space usage. Specifically, the file system checks for total free space of the device and updates the free space counter when a data block is allocated or de-allocated. Metadata blocks in the virtual physical address space are allocated in the same way as the original ext3 file system, thus making use of existing bitmaps.

5.2 Associated Metadata

We include the following items as associated metadata of a data block: 1) the inode number or the logical address of the indirect block that points to the data block, 2) the offset within the inode or the indirect block, 3) the inode generation number, and 4) a timestamp of when the data block was last updated. Items 1 to 3 are used to identify the metadata structure that points to a data block. Item 4 is used during the migration callback process to update the metadata structure with the most up-to-date physical address of a data block.

All the associated metadata is stored in the OOB area of a data block. The total amount of additional status we store in the OOB area is less than 48 bytes, smaller than the typical 128-byte OOB size of 4-KB flash pages. For reliability reasons, we assume that a data page and its OOB area are always written atomically.

5.3 Write

To perform a nameless write, the file system sends the data and the associated metadata of the block to the device. When the device finishes a nameless write and sends back its physical address, the file system updates the inode or the indirect block pointing to it with the new physical address. It also updates the block buffer with the new physical address. In ordered journaling mode, metadata blocks are always written after data blocks have been committed; thus on-disk metadata is always consistent with its data. The file system performs overwrites similarly. The only difference is that overwrites have an existing physical address, which is sent to the device; the device uses this information to invalidate the old data.

5.4 Read

We change two parts of the read operation of data blocks in the physical address space: reading from the page cache and reading from the physical device. To search for a data block in the page cache, we compare the metadata index (e.g., inode number, inode generation number, and block offset) of the block to be read against the metadata associated with the blocks in the page cache. If the buffer is not in the page cache, the file system fetches it from the device using its physical address. The associated metadata of the data block is also sent with the read operation to enable the device to search for remapping entries during device wear leveling (see Section 5.6).

5.5 Free

The current Linux ext3 file system does not support the SSD trim operation. We implemented the ext3 trim operation in a manner similar to ext4. Trim entries are created when the file system deletes a block (named or nameless). A trim entry contains the logical address of a named block or the physical address of a nameless block, the length of the block, its associated metadata, and the address space flag. The file system then adds the trim entry to the current journal transaction. At the end of transaction commit, all trim entries belonging to the transaction are sent to the device. The device locates the block to be deleted using the information contained in the trim operation and invalidates the block.

When a metadata block is deleted, the original ext3 de-allocation process is performed. When a data block is deleted, no de-allocation is performed (i.e., bitmaps are not updated); only the free space counter is updated.

5.6 Wear Leveling with Callbacks

When a nameless-writing device performs wear leveling, it migrates live data to achieve even wear of the device. When such migration happens with data blocks in the physical address space, the file system needs to be informed about the change of their physical addresses. In this section, we describe how the nameless-writing device handles data block migration and how it interacts with the file system to perform *migration callbacks*.

When live nameless data blocks (together with their associated metadata in the OOB area) are migrated during wear leveling, the nameless-writing device creates a mapping from the data block's old physical address to its new physical address and stores it together with its associated metadata in a *migration mapping table* in the device cache. The migration mapping table is used to locate the migrated physical address of a data block for reads and overwrites, which may be sent to the device with the block's old physical address. After the mapping has been added, the old physical address is reclaimed and can be used by future writes.

At the end of a wear-leveling operation, the device sends a migration callback to the file system, which contains all migrated physical addresses and their associated metadata. The file system then uses the associated metadata to locate the metadata pointing to the data block and updates it with the new physical address in a background process. Next, the file system writes changed metadata to the device. When a metadata write finishes, the file system deletes all the callback entries belonging to this metadata block and sends a response to the device, informing it that the migration callback has been processed. Finally, the device deletes the remapping entry when receiving the response of a migration callback.

For migrated metadata blocks, the file system does not need to be informed of the physical address change since it is kept in the virtual address space. Thus, the device does not keep remapping entries or send migration callbacks for metadata blocks.

During the migration callback process, we allow reads and overwrites to the migrated data blocks. When receiving a read or an overwrite during the callback period, the device first looks in the migration mapping table to locate the current physical address of the data block and then performs the request.

Since all remapping entries are stored in the on-device RAM before the file system finishes processing the migration callbacks, we may run out of RAM space if the file system does not respond to callbacks or responds too slowly. In such a case, we simply prohibit future wear-leveling migrations and prevent block wear-out only through garbage collection.

5.7 Reliability Discussion

The changes of the ext3 file system discussed above may cause new reliability issues. In this section, we discuss several reliability issues and our solutions to them.

There are three main reliability issues related to nameless writes. First, we maintain a mapping table in the on-device RAM for the virtual address space. This table needs to be reconstructed each time the device powers on (either after a normal power-off or a crash). Second, the in-memory metadata can be inconsistent with the physical addresses of nameless blocks because of a crash after writing a data block and before updating its metadata block, or because of a crash during wear-leveling callbacks. Finally, crashes can happen during in-place garbage collection, specifically, after reading the live data and before writing them back, which may cause data loss.

We solve the first two problems by using the metadata information maintained in the device OOB area. We store logical addresses with data pages in the virtual address space for reconstructing the logical-to-physical address mapping table. We store associated metadata, as discussed in Section 3.4, with all nameless data. We also store the validity of all flash pages in their OOB area. We maintain an invariant that metadata in the OOB area is always consistent with the data in the flash page by writing the OOB area and the flash page atomically.

We solve the in-place garbage collection reliability problem by requiring the use of a small memory backed by battery or super-capacitor. Notice that the amount of live data we need to hold during a garbage collection operation is no more than the size of an SSD block, typically 256 KB, thus only adding a small monetary cost to the whole device.

The recovery process works as follows. When the device is started, we perform a whole-device scan and read the OOB area of all valid flash pages to reconstruct the mapping table of the virtual address space. If a crash is detected, we perform the following steps. The device sends the associated metadata in the OOB area and the physical addresses of flash pages in the physical address space to the file system. The file system then locates the proper metadata structures. If the physical address in a metadata structure is inconsistent, the file system updates it with the new physical address and adds the metadata write to a dedicated transaction. After all metadata is processed, the file system commits the transaction, at which point the recovery process is finished.

6 Evaluation

In this section, we present our evaluation of nameless writes on an emulated nameless-writing device. Specifically, we focus on studying the following questions:

- What are the space costs of nameless-writing devices compared to other FTLs?

Configuration	Value
SSD Size	4 GB
Page Size	4 KB
Block Size	256 KB
Number of Planes	10
Hybrid Log Block Area	5%
Page Read Latency	25 μs
Page Write Latency	200 μs
Block Erase Latency	1500 μs

Table 2: **SSD Emulator Configuration.**

Image Size	Page	Hybrid	Nameless
328 MB	328 KB	38 KB	2.7 KB
2 GB	2 MB	235 KB	12 KB
10 GB	10 MB	1.1 MB	31 KB
100 GB	100 MB	11 MB	251 KB
400 GB	400 MB	46 MB	1 MB
1 TB	1 GB	118 MB	2.2 MB

Table 3: **FTL Mapping Table Size.** *Mapping table size of page-level, hybrid, and nameless-writing devices with different file system images. The configuration in Table 2 is used.*

- What is the overall performance benefit of nameless-writing devices?
- What is the write performance of nameless-writing devices? How and why is it different from page-level mapping and hybrid mapping FTLs?
- What is the cost of in-place garbage collection and the overhead of wear-leveling callbacks?
- Is crash recovery correct and what are its overheads?

SSD Emulator: We built an SSD emulator which models a multi-plane SSD with garbage collection and wear leveling as a pseudo block device based on David [4]. We implemented three types of FTLs: page-level mapping, hybrid mapping and nameless-writing on top of the PSU objected-oriented SSD simulator codebase [6]. Data is stored in memory to enable quick and accurate emulation. Table 2 describes the configuration we used.

The page-level mapping FTL writes data in a log-structured fashion and schedules in round-robin order across parallel planes. It keeps a mapping for each data page between its logical and physical address. We assume (unrealistically) that this SSD has enough memory to store all page-level mappings. The page-level SSD serves as an upper-bound on performance.

We implemented a hybrid mapping FTL similar to FAST [22], which uses a *log block area* for random data and one sequential log block dedicated for sequential streams. The rest of the device is a *data block area* used to store whole data blocks. The hybrid mapping FTL maintains the page-level mapping of the log block area and the block-level mapping of the data block area.

We implemented a simple garbage collection algorithm that recycles blocks with the least live data in page-level mapping and hybrid mapping FTLs, and a wear-leveling algorithm on all three FTLs that considers a block's remaining erase cycles and its data temperature during wear leveling similar to a previous wear-leveling algorithm [2].

System Setup: We implemented the emulated nameless-write device and the nameless-writing ext3 file system on a 64-bit Linux 2.6.33 kernel. The page-level mapping and the hybrid mapping SSD emulators are built on an unmodified 64-bit Linux 2.6.33 kernel. All experiments are performed on a 2.5 GHz Intel Quad Core CPU with 8 GB memory.

6.1 SSD Memory Consumption

We first study the space cost of mapping tables used by different SSD FTLs: nameless-writing, page-level mapping, and hybrid mapping. The mapping table size of page-level and hybrid FTLs is calculated based on the total size of the device, its block size, and its log block area size (for hybrid mapping). A nameless-writing device keeps a mapping table for the entire file system's virtual address space. Since we map all metadata to the virtual block space in our nameless-writing implementation, the mapping table size of the nameless-writing device is dependent on the metadata size of the file system image. We use Impressions [3] to create typical file system images of sizes up to 1 TB and calculate their metadata sizes.

Figure 3 shows the mapping table sizes of the three FTLs with different file system images produced by Impressions. Unsurprisingly, the page-level mapping has the highest mapping table space cost. The hybrid mapping has a moderate space cost; however, its mapping table size is still quite large: over 100 MB for a 1-TB device. The nameless mapping table has the lowest space cost; even for a 1-TB device, its mapping table uses less than 3 MB of space for typical file systems, reducing both cost and power usage.

6.2 Application Performance

We now present the overall application performance of nameless-writing, page-level mapping and hybrid mapping FTLs with macro-benchmarks. We use varmail, fileserver, and webserver from the filebench suite [29].

Figure 2 shows the throughput of these benchmarks. We see that both page-level mapping and nameless-writing FTLs perform better than the hybrid mapping FTL with varmail and fileserver. These benchmarks contain 90.8% and 70.6% random writes, respectively. As we will see later in this section, the hybrid mapping FTL performs well with sequential writes and poorly with random writes. Thus, its throughput for these two benchmarks

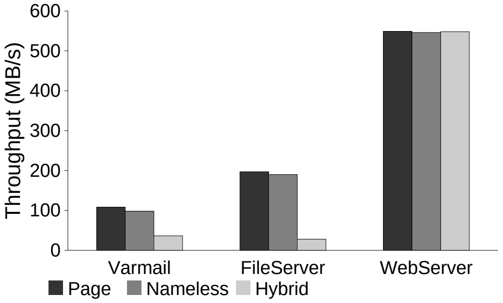

Figure 2: **Throughput of Filebench.** *Throughput of varmail, fileserver, and webmail macro-benchmarks with page-level, nameless-writing, and hybrid FTLs.*

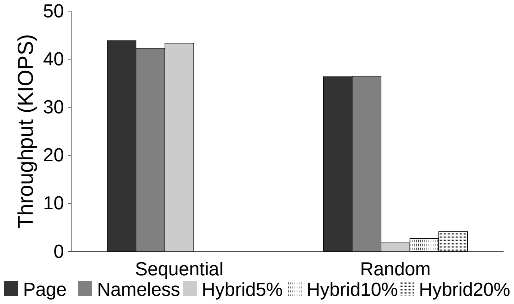

Figure 3: **Sequential and Random Write Throughput.** *Throughput of sequential writes and sustained 4-KB random writes. Random writes are performed over a 2-GB range.*

is worse than the other two FTLs. For webserver, all three FTLs deliver similar performance, since it contains only 3.8% random writes. We see a small overhead of the nameless-writing FTL as compared to the page-level mapping FTL with all benchmarks, which we will discuss in detail in Sections 6.5 and 6.6.

In summary, we demonstrate that the nameless-writing device achieves excellent performance, roughly on par with the costly page-level approach, which serves as an upper-bound on performance.

6.3 Basic Write Performance

Write performance of flash-based SSDs is known to be much worse than read performance, with random writes being the performance bottleneck. Nameless writes aim to improve write performance of such devices by giving the device more data-placement freedom. We evaluate the basic write performance of our emulated nameless-writing device in this section. Figure 3 shows the throughput of sequential writes and sustained 4-KB random writes with page-level mapping, hybrid mapping, and nameless-writing FTLs.

First, we find that the emulated hybrid-mapping device has a sequential throughput of 169 MB/s and a sustained 4-KB random write throughput of 2,830 IOPS. A widely used real middle-end SSD has sequential throughput of up to 70 MB/s and random throughput of up to 3,300 IOPS [17]. Although the write performance of our emulator does not match this real SSD exactly, it is still in the ballpark of actual SSD performance, and thus useful in our study. The goal of our hybrid-mapping emulator is not to model one particular SSD perfectly but to provide insight into the fundamental problems of hybrid-mapped SSDs as compared to page-mapped and nameless SSDs.

Second, the random write throughput of page-level mapping and nameless-writing FTLs is close to their sequential write throughput, because both FTLs allocate data in a log-structured fashion, making random writes behave like sequential writes. The overhead of random writes with these two FTLs comes from their garbage collection process. Since whole blocks can be erased when they are overwritten in sequential order, garbage collection has the lowest cost with sequential writes. By contrast, garbage collection of random data may incur the cost of live data migration.

Third, we notice that the random write throughput of the hybrid mapping FTL is significantly lower than that of the other FTLs and its own sequential write throughput. The poor random write performance of the hybrid mapping FTL results from the costly full-merge operation and its corresponding garbage collection process [16]. Full merges are required each time a log block is filled with random writes, thus a dominating cost for random writes.

One way to improve the random write performance of hybrid-mapped SSDs is to over-provision more log block space. To explore that, we vary the size of the log block area with the hybrid mapping FTL from 5% to 20% of the whole device and found that random write throughput gets higher as the size of the log block area increases. However, only the data block area reflects the effective size of the device, while the log block area is part of device over-provisioning. Therefore, hybrid-mapped SSDs often sacrifice device space cost for better random write performance. Moreover, the hybrid mapping table size increases with higher log block space, requiring larger on-device RAM. Nameless writes achieve significantly better random write performance with no additional over-provisioning or RAM space.

Finally, Figure 3 shows that the nameless-writing FTL has low overhead as compared to the page-level mapping FTL with sequential and random writes. We explain this result in more detail in Section 6.5 and 6.6.

6.4 A Closer Look at Random Writes

A previous study [16] and our study in the last section show that random writes are the major performance bottleneck of flash-based devices. We now study two subtle yet fundamental questions: do nameless-writing devices

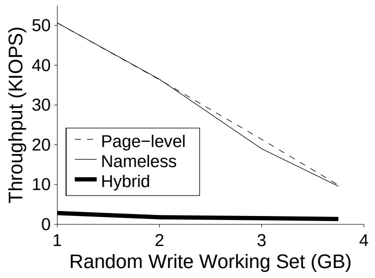

Figure 4: **Random Write Throughput.** *Throughput of sustained 4-KB random writes over different working set sizes with page-level, nameless, and hybrid FTLs.*

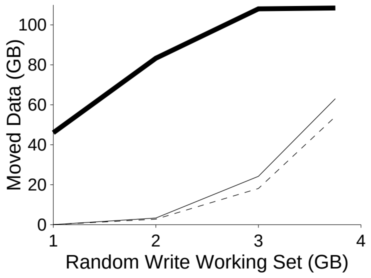

Figure 5: **Migrated Live Data.** *Amount of migrated live data during garbage collection of random writes with different working set sizes with page-level, nameless, and hybrid FTLs.*

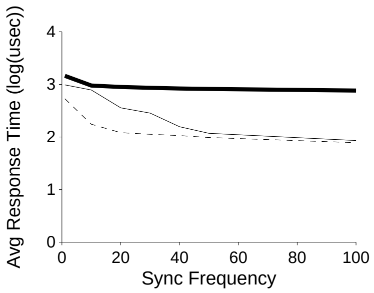

Figure 6: **Average Response Time of Synchronous Random Writes.** *4-KB random writes in a 2-GB file. Sync frequency represents the number of writes we issue before calling an fsync.*

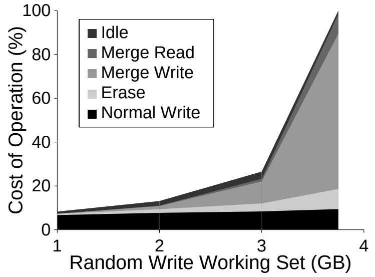

Figure 7: **Page-Level FTL Utilization.** *Break down of device utilization with the page-level FTL under random writes of different ranges.*

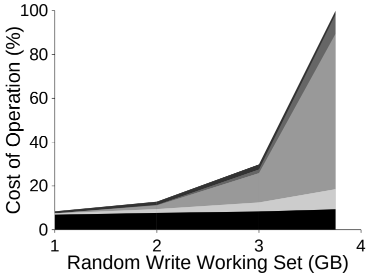

Figure 8: **Nameless FTL Utilization.** *Break down of device utilization with the nameless FTL under random writes of different ranges.*

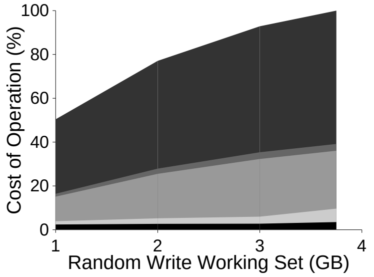

Figure 9: **Hybrid FTL Utilization.** *Break down of device utilization with the hybrid FTL under random writes of different ranges.*

perform well with different kinds of random-write workloads, and why do they outperform hybrid devices.

To answer the first question, we study the effect of working set size on random writes. We create files of different sizes and perform sustained 4-KB random writes in each file to model different working set sizes. Figure 4 shows the throughput of random writes over different file sizes with all three FTLs. We find that the working set size has a large effect on random write performance of nameless-writing and page-level mapping FTLs. The random write throughput of these FTLs drops as the working set size increases. When random writes are performed over a small working set, they will be overwritten in full when the device fills and garbage collection is triggered. In such cases, there is a higher chance of finding blocks that are filled with invalid data and can be erased with no need to rewrite live data, thus lowering the cost of garbage collection. In contrast, when random writes are performed over a large working set, garbage collection has a higher cost since blocks contain more live data, which must be rewritten before erasing a block.

To further understand the increasing cost of random writes as the working set increases, we plot the total amount of live data migrated during garbage collection (Figure 5) of random writes over different working set sizes with all three FTLs. This graph shows that as the working set size of random writes increases, more live data is migrated during garbage collection for these FTLs, resulting in a higher garbage collection cost and worse random write performance.

Comparing the page-level mapping FTL and the nameless-writing FTL, we find that nameless-writing has slightly higher overhead when the working set size is high. This overhead is due to the cost of in-place garbage collection when there is wasted space in the recycled block. We will study this overhead in details in the next section.

We now study the second question to further understand the cost of random writes with different FTLs. We break down the device utilization into regular writes, block erases, writes during merging, reads during merging, and device idle time. Figures 7, 8, and 9 show the stack plot of these costs over all three FTLs. For page-level mapping and nameless-writing FTLs, we see that the major cost comes from regular writes when random writes are performed over a small working set. When the working set increases, the cost of merge writes and erases increases

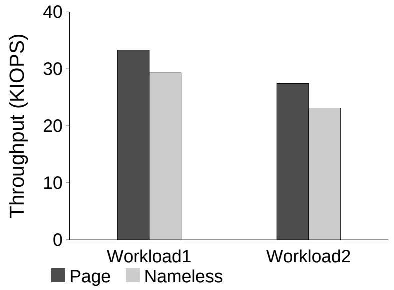

Figure 10: **Write Throughput with Wear leveling.** *Throughput of biased sequential writes with wear leveling under page-level and nameless FTLs.*

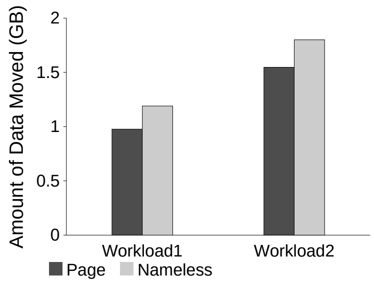

Figure 11: **Migrated Live Data during Wear Leveling.** *Amount of migrated live data during wear leveling under page-level and nameless FTLs.*

	Metadata	RemapTbl
Workload1	2.02 MB	321 KB
Workload2	5.09 MB	322 KB

Figure 12: **Wear leveling Callback Overhead.** *Amount of additional metadata writes because of migration callbacks and maximal remapping table size during wear leveling with the nameless-writing FTL.*

and becomes the major cost. For the hybrid mapping FTL, the major cost of random writes comes from migrating live data and idle time during merging for all working set sizes. When the hybrid mapping FTL performs a full merge, it reads and writes pages from different planes, thus creating idle time on each plane.

In summary, we demonstrate that random write throughput of the nameless-writing FTL is close to that of the page-level mapping FTL and is significantly better than the hybrid mapping FTL, mainly because of the costly merges the hybrid mapping FTL performs for random writes. We also found that both nameless-writing and page-level mapping FTLs achieve better random write throughput when the working set is relatively small because of a lower garbage collection cost.

6.5 In-place Garbage Collection Overhead

The performance overhead of a nameless-writing device may come from two different device responsibilities: garbage collection and wear leveling. We study the overhead of in-place garbage collection in this section and wear-leveling overhead in the next section.

Our implementation of the nameless-writing device uses an in-place merge to perform garbage collection. As explained in Section 4.2, when there are no waiting writes on the device, we may waste the space that has been recently garbage collected. We use synchronous random writes to study this overhead. We vary the frequency of calling *fsync* to control the amount of waiting writes on the device; when the sync frequency is high, there are fewer waiting writes on the device queue. Figure 6 shows the average response time of 4-KB random writes with different sync frequencies under page-level mapping, nameless-writing, and hybrid mapping FTLs. We find that when sync frequency is high, the nameless-writing device has a larger overhead compared to page-level mapping. This overhead is due to the lack of waiting writes on the device to fill garbage-collected space. However, we see that the average response time of the nameless-writing FTL is still lower than that of the hybrid mapping FTL, since response time is worse when the hybrid FTL performs full-merge with synchronous random writes.

6.6 Wear-leveling Callback Overhead

Finally, we study the overhead of wear leveling in a nameless-writing device. To perform wear-leveling experiments, we reduce the lifetime of SSD blocks to 50 erase cycles. We set the threshold of triggering wear leveling to be 75% of the maximal block lifetime, and set blocks that are under 90% of the average block remaining time to be candidates for wear leveling.

We create two workloads to model different data temperature and SSD wear: a workload that first writes 3.5-GB data in sequential order and then overwrites the first 500-MB area 40 times (Workload 1), and a workload that overwrites the first 1-GB area 40 times (Workload 2). Workload 2 has more hot data and triggers more wear leveling. We compare the throughput of these workloads with page-level mapping and nameless-writing FTLs in Figure 10. The throughput of Workload 2 is worse than that of Workload 1 because of its more frequent wear-leveling operation. Nonetheless, the performance of the nameless-writing FTL with both workloads has less than 9% overhead.

We then plot the amount of migrated live data during wear leveling with both FTLs in Figure 11. As expected, Workload 2 produces more wear-leveling migration traffic. Comparing page-level mapping to nameless-writing FTLs, we find that the nameless-writing FTL migrates more live data. When the nameless-writing FTL performs in-place garbage collection, it generates more migrated live data, as shown in Figure 5. Therefore, more erases are caused by garbage collection with the nameless-writing FTL, resulting in more wear-leveling invocation and more wear-leveling migration traffic.

Migrating live nameless data in a nameless-writing device creates callback traffic and additional metadata writes. Wear leveling in a nameless-writing device also adds a space overhead when it stores the remapping table for migrated data. We show the amount of additional metadata writes and the maximal size of the remapping table of a nameless-writing device in Figure 12. We find both overheads to be low with the nameless-writing device: an addition of less than 6 MB metadata writes and a space cost of less than 350 KB.

In summary, we find that both the garbage-collection and wear-leveling overheads caused by nameless writes are low. Since wear leveling is not a frequent operation and is often scheduled in system idle periods, we expect both performance and space overheads of a nameless-writing device to be even lower in real systems.

6.7 Reliability

To determine the correctness of our reliability solution, we inject crashes in the following points: 1) after writing a data block and its metadata block, 2) after writing a data block and before updating its metadata block, 3) after writing a data block and updating its metadata block but before committing the metadata block, and 4) after the device migrates a data block because of wear leveling and before the file system processes the migration callback. In all cases, we successfully recover the system to a consistent state that correctly reflects all written data blocks and their metadata.

Our results also show that the overhead of our crash recovery process is relatively small: from 0.4 to 6 seconds, depending on the amount of inconsistent metadata after crash. With more inconsistent metadata, the overhead of recovery is higher.

7 Related Work

A large body of work on flash-based SSD FTLs and file systems that manage them has been proposed in recent years [11, 14, 16, 19, 21, 22, 25, 33]. In this section, we discuss the two research projects that are most related to nameless writes.

Range writes [5] use an approach similar to nameless writes. Range writes were proposed to improve hard disk performance by letting the file system specify a range of addresses and letting the device pick the final physical address of a write. Instead of a range of addresses, nameless writes are not specified with any addresses, thus obviating file system allocation and moving allocation responsibility to the device. Problems such as updating metadata after writes in range writes also arise in nameless writes. We propose a segmented address space to lessen the overhead and the complexity of such an update process. Another difference is that nameless writes target devices that need to maintain control of data placement, such as wear leveling in flash-based devices. Range writes target traditional hard disks that do not have such responsibilities. Data placement with flash-based devices is also less restricted than traditional hard disks, since flash-based memory has uniform access latency regardless of its location.

The poor random write performance of hybrid FTLs has drawn attention from researchers in recent years. The demand-based Flash Translation Layer (DFTL) was proposed to address this problem by maintaining a page-level mapping table and writing data in a log-structured fashion [16]. DFTL stores its page-level mapping table on the device and keeps a small portion of the mapping table in the device cache based on workload temporal locality. However, for workloads that have a bigger working set than the device cache, swapping the cached mapping table with the on-device mapping table structure can be costly. There is also a space overhead to store the entire page-level mapping table on device. We use a log-structured write order similar to DFTL to maximize the device's sequential writing capability. However, the need for a device-level mapping table is obviated with nameless writes. Indirection is maintained only for the virtual address space, which as we show, requires a small space cost and can fit in the device cache with typical file system images. Thus, we do not pay the space cost of storing the large page-level mapping table in the device or the performance overhead of swapping mapping table entries.

8 Conclusions and Future Work

In this paper, we introduced nameless writes, a new write interface built to reduce the inherent costs of indirection. Through the implementation of nameless writes on the Linux ext3 file system and an emulated nameless-writing device, we demonstrate how to port a file system to use nameless writes. Through extensive evaluations, we show the great advantage of nameless writes: greatly reduced space costs and improved random-write performance.

Porting other types of file systems to use nameless writes would be interesting and is a part of our future work. Here, we give a brief discussion about these file systems and the challenges we foresee in changing them to use nameless writes.

Linux ext2: The Linux ext2 file system is similar to the ext3 file system except that it has no journaling. While we rely on the ordered journal mode to provide a natural ordering for the metadata update process of nameless writes in ext3, we need to introduce an ordering on the ext2 file system. Our initial implementation of nameless-writing ext2 shows that one possible method to enforce such an ordering is to defer metadata writes until all the ongoing data writes belonging to them have finished.

Copy-On-Write File Systems and Snapshots: As an alternative to journaling, *copy-on-write* (COW) file systems always write out updates to new free space; when all of those updates have reached the disk, a root structure is updated to point at the new structures, and thus include them in the state of the file system. COW file systems thus map naturally to nameless writes. All writes to free space are mapped into the physical segment and issued namelessly; the root structure is mapped into the virtual segment. The write ordering is not affected, as COW file systems all must wait for the COW writes to complete before issuing a write to the root structure anyway.

One problem with COW file systems or other file systems that support snapshots or versions is that multiple metadata structures can point to the same data block, which may result in a large amount of associated metadata. We can use file system intrinsic back references, such as those in btrfs, or structures like *Backlog* [23] to represent associated metadata. Another problem is that multiple metadata blocks need to be updated after a nameless write. One possible way to control the number of metadata updates is to reduce the amount of metadata included in the virtual address space.

Extent-Based File Systems: One final type of file systems worth considering are *extent-based* file systems, such as Linux btrfs and ext4, where contiguous regions of a file are pointed to via (pointer, length) pairs instead of a single pointer per fixed-sized block. Modifying an extent-based file system to use nameless writes would require a bit of work; as nameless writes of data are issued, the file system would not (yet) know if the data blocks will form one extent or many. Thus, only when the writes complete will the file system be able to determine the outcome. Later writes would not likely be located nearby, and thus to minimize the number of extents, updates should be issued at a single time. Extents also hint at the possibility of a new interface for nameless writes. Specifically it might be useful to provide an interface to *reserve* a larger contiguous region on the device; doing so would enable the file system to ensure that a large file was placed contiguously in physical space, and thus affords a highly compact extent-based representation. We plan to look into such enhancements in the future.

Acknowledgment

We thank the anonymous reviewers and Jason Flinn (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank the members of the ADSL research group for their insightful comments.

This material is based upon work supported by the National Science Foundation under the following grant: NSF CCF-0937959, as well as by generous donations from Google, NetApp, and Samsung.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[2] N. Agarwal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[3] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, February 2009.

[4] N. Agrawal, L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Emulating Goliath Storage Systems with David. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST '11)*, San Jose, California, February 2011.

[5] A. Anand, S. Sen, A. Krioukov, F. Popovici, A. Akella, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Banerjee. Avoiding File System Micromanagement with Range Writes. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[6] B. Tauras, Y. Kim, and A. Gupta. PSU Objected-Oriented Flash based SSD simulator. http://csl.cse.psu.edu/?q=node/321.

[7] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[8] S. Boboila and P. Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[9] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[10] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[11] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System Software for Flash Memory: A Survey. In *Proceedings of thei 5th International Conference on Embedded and Ubiquitous Computing (EUC '06)*, pages 394–404, August 2006.

[12] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[13] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 237–252, San Francisco, California, January 1992.

[14] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37:138–163, June 2005.

[15] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of MICRO-42*, New York, New York, December 2009.

[16] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 43th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 229–240, Washington, DC, March 2009.

[17] Intel Corporation. Intel X25-M Mainstream SATA Solid-State Drives. `ftp://download.intel.com/design/flash/NAND/mainstream/mainstream-sata-s%sd-datasheet.pdf`.

[18] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee. A Group-based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '07)*, October 2007.

[19] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-Based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT '08)*, Seoul, Korea, August 2006.

[20] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory Based File System. In *Proceedings of the USENIX 1995 Winter Technical Conference*, New Orleans, Louisiana, January 1995.

[21] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *In Proceedings of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008)*, February 2008.

[22] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. *IEEE Transactions on Embedded Computing Systems*, 6, 2007.

[23] P. Macko, M. Seltzer, and K. A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[24] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 238–251, Saint-Malo, France, October 1997.

[25] A. One. YAFFS: Yet Another Flash File System, 2002. `http://www.yaffs.net/`.

[26] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference*

on the Management of Data (SIGMOD '88), pages 109–116, Chicago, Illinois, June 1988.

[27] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[28] D. Spinellis. Another Level of Indirection. In A. Oram and G. Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 17, pages 279–291. O'Reilly and Associates, 2007.

[29] Sun Microsystems. Solaris Internals: FileBench. `http://www.solarisinternals.com/wiki/index.php/FileBench`.

[30] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[31] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[32] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[33] D. Woodhouse. JFFS2: The Journalling Flash File System, Version 2, 2001. `http://sources.redhat.com/jffs2/jffs2`.