Removing The Costs Of Indirection in Flash-based SSDs with Nameless Writes

Andrea C. Arpaci-Dusseau*, Remzi H. Arpaci-Dusseau*, and Vijayan Prabhakaran+
University of Wisconsin–Madison and Microsoft Research+*

Abstract

We present *nameless writes*, a new interface that obviates the need for indirection in modern solid-state storage devices (SSDs). Nameless writes allow the device to pick the location of a write and only then inform the client above of the decision. Doing so keeps control of block allocation decisions in the device, thus enabling it to perform important tasks such as wear-leveling, while removing the need for large and costly indirection tables. We discuss the proposed interface as well as the requisite device and file-system support.

1 Introduction

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

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

Indirection has many other benefits. In AutoRAID, for example, a level of indirection allows the system to keep "active" blocks to mirrored storage for performance reasons, and move "inactive" blocks to RAID to increase effective capacity [14]. Indirection further allows AutoRAID to write blocks in log-structured fashion, hence overcoming the RAID small-update problem.

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.

FTLs use indirection for two reasons: first, to transform the erase/program cycle into the more typical write-based interface via copy-on-write techniques, and second, to implement *wear leveling*, which is critical to increasing SSD lifetime. Specifically, because a flash block becomes unusable after a certain number of erase-program cycles (10,000 or 100,000 cycles according to manufacturers, but perhaps more in practice [2, 7]), such indirection is needed to spread the write load across flashes evenly and thus ensuring that no particularly popular block will cause the device to fail sooner than expected.

Unfortunately, indirection such as found in many FTLs comes at a high cost, which can manifest as performance costs or space overheads or both. If the FTL can flexibly map each virtual *page* in its address space (assuming a typical page size of 2KB), an incredibly large indirection table is required. For example, a 1-TB SSD needs 2 GB of 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 (which usually is SRAM) into an SSD is prohibitive.

Because of this high cost, most SSDs instead do not offer a fully flexible per-page mapping. A simple approach could provide only a pointer per *block* of the SSD (a block typically containing 64 or 128 2-KB pages), which reduces overheads by the ratio of block size to page size (our 1-TB example above is thus reduced to 32 MB, which, although more reasonable, is not stellar). However, as clearly articulated by Gupta et al. [8], block-level mappings have high costs due to excessive garbage collection.

As a result, most FTLs today use a hybrid FTL, which maps most data via block pointers and updates to the device via page pointers. Hybrid approaches keep space overheads low while avoiding the high costs of garbage collection, at the cost of additional device complexity. Unfortunately, garbage collection can still be costly, thus reducing the performance of the SSD, sometimes noticeably [8]. Regardless of the approach, FTL indirection incurs a significant cost; as SSDs scale, even schemes mostly based on block pointers will become infeasible.

In this paper, instead of advocating a subtly different indirection scheme within an FTL, we suggest a new approach which removes the need for indirection altogether. Our approach to *de-indirection* centers around a new type of write to the storage system which we refer to as a *nameless write*. Unlike most writes, which specify both the *data* to write as well as a *name* (usually in the form of a simple address), nameless writes simply pass the data

to the device. The device is free to choose any underlying physical block for the data; after the device *names* the block, it informs the client of its choice, which the client then can use to record the name for future reads.

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 of the page to the client above (e.g., the file system), the need for indirection is avoided, this reducing the cost of the SSD as well as simplifying its internal structure dramatically.

Nameless writes remove the costs of indirection without giving away the primary responsibility any SSD manufacturer would like to have: wear leveling. If an SSD simply exported 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. With nameless writes, no such failure mode exists, as the device does not allow a named write to a physical block. 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.

Finally, nameless writes work well with devices that offer true random access. Although a series of nameless writes may often end up contiguous, if they do not, it is less of a problem on a medium such as flash. Note an added advantage: the physical block layout is immediately *visible* to the file system above. We believe this information is useful in a number of contexts (e.g., security, reliability) and plan to investigate further in the future.

Of course, nameless writes are not without their difficulties. Most importantly, nameless writes demand certain properties from file systems above. Specifically, we believe that copy-on-write (or shadow-paging) file systems are more readily adopted to an SSD with nameless writes, though perhaps other file systems can also be "ported" to use nameless writes. Second, simply naming each write at the device-level instead of at the file-system level (a process we refer to as *late binding*) is not enough, as the device may wish to later migrate a block to another location during garbage collection. Thus, a *renaming callback* must also be apart of the new device interface, which allows the device to inform the file system of a migration of a particular block to a new physical location. In this paper, we discuss these and other important issues, presenting our initial thoughts as a starting point for further discussion.

Read
down: `address, length`
up: `status, data`

Nameless Write
down: `data, length`
up: `status, resulting target address(es)`

Named Write
down: `data, address, length`
up: `status`

Free (or Trim)
down: `address, length`
up: `status`

Reclaim [Callback]
up: `address, data(?)`

Table 1: **New Device Interface: Version 1.** *The table shows the new interface to storage as dictated by nameless writes.*

2 The New Device Interface

In this section, we discuss the new device interface to support nameless writes. Table 1 summarizes the interface, which we now present in more detail.

The interface to *read* a block remains unchanged: the client sends an address and a length down to the device; the device replies some time later with a status message (e.g., success or failure), and if successful, the requested data. Note that the name is essentially a physical address.

Next is the *nameless write*, which is different from a typical write in two salient ways. First, the write does not specify a target address (i.e., a name); this essential component of the interface is key to letting the device select the physical location without control from above. Second, in response, the device writes the data to the device, and returns the name (i.e., the physical address) to the calling client (as well as a status indicator).

Note that in this incarnation of the interface, a single write is not guaranteed to be mapped to contiguous blocks. Thus, a "large" write may yield a number of resulting names and lengths, as specified in the returned values. However, it is generally best if the file system issues writes in multiple of device pages (i.e., 2 KB), as each write can thus be easily redirected to a meaningful physical location without waste.

Because a nameless write is an allocating operation, the device needs to also be informed of a deallocation of the block by higher-level software. Most modern devices refer to this interface as the *free* or *trim* command. Once a

block has been freed/trimmed, the device can re-use it.

We also include a *named write*, which is just a typical write as one sees in today's storage systems. As we will see when discussing file system implementation below, named writes are needed in the few cases when the updated block must have a known location, so that the file system can find it later (say, when mounting). This interface should only be used sparingly; indeed, using it primarily could revert the SSD with little demand for remapping back into a typical remapping SSD (perhaps providing an upgrade path?). One small complication: because nameless writes and reads generally deal with physical addresses, one needs a way to specify the address of a named write. We imagine that the first so many (thousand? million?) addresses in the address space will be thus reserved.

Finally, one of the more novel aspects of the interface is the *reclaiming callback*, which allows the device to inform the file system of its need to reclaim a physical block. This upcall allows the device to tell the client above that it needs to write to the physical block again (for the purpose of wear leveling), and thus informs the client which address it wishes to reclaim (and perhaps even hands the data to the client). We explain further via example.

Imagine a scenario where a file system writes data to the device and said data is bound to a specific physical address. Unfortunately, the block that the data is within contains data which soon becomes cold (but is all still live); in this case, the block is fully live but not being written to, and thus will soon become the target of the wear-leveling desires of the device. If the device had an indirection map, it could simply move the block and update the map; however, here we assume that one generally does not have such a map. Thus, with nameless writes, the device instead *informs* the client (i.e., the file system) that it wishes to reclaim a particular physical block. At this point, it becomes the file system's responsibility to rewrite the block to a new location and then eventually trim the old block, thus allowing the device to reclaim it.

2.1 Alternate Interfaces

We came to this first proposed interfaces after considering a number of alternatives; thus, this interface represents our current "best", although it is by no means final and absolutely represents a place where we hope for feedback through the workshop.

One aspect of the interface where we engaged in much discussion was around the nature of the callback interface. Our current approach we think is simplest, but many other possibilities exist. For example, one idea of ours built upon the type-safe disks work of Sivathanu et al. [13]. In this variant of type-safe disks, updates were presented to the device in such a way that the device knew which entities were pointers to other blocks; this made it quite simple for the reclaiming callback to not only inform the client above of a reclaim, but also of the location of the blocks that point to the reclaimed block.

Another variant of the callback was more forceful, not only telling the file system of the desire to move a block but rather already moving it and then telling the file system the result. This may indeed be a reasonable approach, but we felt forced the file system to act too quickly upon the request.

3 Device Implications

We now discuss device-side implications of our current interface proposal. Overall, the device is likely to be somewhat simpler as a result of our changes. Instead of a complex hybrid page/block remapping FTL, a leaner, simpler, less costly device arises.

Because the device handles allocation (on writes), and freeing (on trims), clearly the device must track block liveness. The simplest data structure to support such tracking is a bitmap (used as far back as FFS [11] if not earlier), but more compact and yet fully-functional data structures to track free-space certainly exist (e.g., extent maps as found in XFS and ZFS). However, this is no different that the status quo in SSDs today, which need to track liveness information to perform proper garbage collection.

The new device must also track wear information. However, tracking such usage information is also no different than before and thus exacts a similar overhead in traditional SSDs.

The device must include a small amount of remapping support for named writes, which is similar to remapping support current FTLs provide but much smaller in scale. The device may further decide to remap some "physical" blocks that it has reclaimed via callback, but which the file system has yet to trim. In those cases, the remapping information is only needed until the trim at which point the file system will have written the data to another location.

Finally, the major new interaction for the device is the implementation of an upcall to inform the client that the device wishes to reclaim a block. There are many possible approaches to implementing the callback, including via a simple stand-alone interrupt, or more likely as extra information piggybacked onto some other request. It may particularly be useful to inform the file system that a block that has just been *read* is a good candidate to reclaim; because it just read the block, the file system must already have in memory the requisite information to write the block again, and thus a timely re-write of the block may be particularly painless at that time.

4 File System Implications

It is likely that the greatest cost in supporting nameless writes is borne by the file system. We now discuss how a file system can be designed with nameless writes in mind.

4.1 Basic Organization: COW

We believe the most straightforward file system to "port" to use nameless writes is a file system that employs a *copy-on-write (COW)* or what database researchers would call a *shadow paging* approach. A COW file system essentially never overwrites data in place. Rather, an update, which is comprised of a number of blocks, is written to a completely new location on the persistent store; only a final update to a "root" block is in-place and must be done carefully to avoid problems in the case of failure during said writes. Examples of this type of file system include Sun's ZFS [3], NetApp's WAFL [10], and even the confusingly-named log-structured file system (LFS) [12].

Let us illustrate how an update works in such a system, and then discuss how said update would work with nameless writes. Imagine we have a root block of the file system (similar to the "uberblock" in ZFS, or the "checkpoint region" in LFS) which tells us the physical locations of each inode in the system. Each inode then tells us the location of each block of each inode, and so forth. Imagine that we are writing out a new file F; this means we will need to write out a data block D, an inode I (which refers to the location of D), and a new version of the root block R (which refers to the location of I).

The file system will issue the writes in bottom-up fashion. First, a nameless write of D will be issued. Assume the device allocates physical block 1000 to D and writes it; the device then returns 1000 as the bound name. The file system then updates the in-memory inode I with 1000 as the pointer to D and issues a nameless write of I; the device writes I to 1001 and returns that address as well. Finally, the file system updates R with the mapping, and then commits R to a known location via a named (not nameless) write. LFS alternated between two checkpoint regions in case a crash occurred during the update of one; imagine a similar approach here.

In this manner, most writes to the device can use nameless writes quite naturally. Only at the top level, when we wish to commit a block to a known location so that it may be found later (during mount) must we use a named write; the bulk of I/O is nameless and thus needs no remapping.

This approach does have some performance differences as compared to a typical COW file system. Specifically, it orders writes up the file system tree, whereas a typical COW file system can issue all writes at once (except the write to the root). However, there is still plenty of concurrency available (e.g., different updates to different files have no dependency ordering) and thus we do not expect major performance problems.

4.2 Crash Recovery

One difficult issue that the device must now handle is crash recovery, i.e., what to do when a series of blocks have been allocated at device level but the file system now thinks them free because of a crash. We refer to this problem as an "allocation leak", as the file system's behavior led the device to mistakenly allocate a block that then goes unused. From our example above, this could occur when block D has been written out but a crash occurs before I or R are written.

From the perspective of the (COW) file system, a crash during an update is not a problem because it simply loses all the updates that were on-going; however, any writes that occurred during the update the device now thinks of as "allocated" and thus must be told to "free" them, otherwise the device may forever think them allocated when in fact they are not in use.

We note that allocation leaks occur in current systems as well as with nameless writes. However, because nameless-writing devices have a reclaim callback (discussed further below) which lets the file system know which blocks to free, it is in this sense made easier with nameless writes; the file system may thus occasionally see a reclaim request for a block it thinks free, at which point it should simply trim the block.

4.3 Dealing With Callbacks

Another major challenge a file system on nameless writes must handle is what to do when a callback informs it that the device wishes to reclaim a particular physical block. The natural response is to immediately rewrite the block to a new location, but first there is a problem to overcome: which block points to this block? Only by knowing this can the file system rewrite it and then trim the block for the device to freely reuse.

We thus believe that a nameless-writing file system will likely contain additional data structures to help it find blocks that the device told it to stop using. Specifically, the file system might wish to use some kind of per-inode bloom filter to provide a quick answer on whether to scan all the block pointers to determine whether a file owns a specific block (particularly useful for big files). Further, a complete file system scan for integrity purposes (such as that done by ZFS) should likely incorporate block reclamation into its priorities, thus helping the device reclaim blocks as needed.

4.4 Other File Systems

Beyond copy-on-write, there are other update strategies for modern file systems, including *journaling* [9] (or write-ahead logging) and *soft updates* [5]. Of the two, soft updates seems to more naturally fit into the update sequence that is natural to nameless writes; however, for both types of file systems, more consideration is required.

5 Related Work

Most similar to this proposal for nameless writes is our own work on *range writes*, which proposes a less radical variant of the write interface [1]. With range writes, a file system presents the underlying storage with a set of possible addresses (instead of just one), and thus allows the device to pick the one most suitable given current low-level information (e.g., the exact position of the disk head). In that work it is shown that under certain workloads, such freedom can greatly improve performance. Range writes, however, still give the file system (or other client of the device) ultimate control over block placement, e.g., a client could simply specify a range of just one block. Because of this fact, range writes are not flexible enough to support the demands of modern wear-leveling SSDs and thus not suitable for our purposes here.

Earlier proposals in object-based disks (OSDs) [6] bear some similarities to our approach. However, OSDs by their very nature imply a great deal of metadata overhead (as they must manage the detailed layout of each object), and thus exacerbate the very problem nameless writes attempt to solve. One particular proposal within this space suggests a new write interface without an address [4] to improve performance, and thus represents an early form of nameless writes. Without a paired rewriting callback, such an interface is insufficient to solve the problems presented in this paper.

In other work on improving Flash FTLs, Gupta et al. show how to build a more flexible and performant page-level mapping FTL by taking advantage of temporal locality common within many workloads. Such an approach thus only keeps part of the full mapping in SRAM, putting the "inactive" mappings on flash and keeping them there until needed. Unfortunately, an unexpected workload can cause trouble for this approach, and, in the worst case, lead to two I/O operations on every read, thus halving performance. Indeed, any approach which keeps part of the mapping table on flash will have this property, and thus may not be suitable in circumstances where the design assumptions do not mesh with workload realities.

6 Conclusions

The problem we put forth herein is real: as SSDs grow in size, large remapping tables will become an increasing cost and performance problem. Nameless writes present one possible solution path; by providing a new interface to storage, one level of indirection can be de-indirected, leaving the benefits of indirection in place without its high costs. Much work remains to be done; what we present here is just a first step towards what we hope will be a fruitful path towards leaner, simpler SSD-based storage.

7 Acknowledgments

We thank the all reviewers for their feedback and comments. This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CNS-0834392, CCF-0811697, CCF-0811697, CCF-0937959, as well as by generous donations from NetApp, Sun Microsystems, and Google. 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] Ashok Anand, Sayandeep Sen, Andrew Krioukov, Florentina Popovici, Aditya Akella, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Suman Banerjee. Avoiding File System Micromanagement with Range Writes. In *OSDI '08*, San Diego, CA, December 2008.

[2] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *FAST '10*, San Jose, California, February 2010.

[3] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[4] Zvi Dubitzky, Israel Gold, Ealan Henis, Julian Satran, and Dafna Sheinwald. DSF – Data Sharing Facility. Technical Report H-0141, IBM Research Division, Haifa Research Laboratory, October 2002.

[5] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *OSDI '94*, pages 49–60, Monterey, CA, November 1994.

[6] Garth A. Gibson, David Rochberg, Jim Zelenka, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, and Erik Riedel. File server scaling with network-attached secure disks. In *SIGMETRICS '97*, pages 272–284, Seattle, Washington, June 1997.

[7] 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 *MICRO-42*, New York, NY, December 2009.

[8] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS XIV*, pages 229–240, Washington, DC, March 2009.

[9] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, Austin, TX, November 1987.

[10] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *USENIX Winter '94*, San Francisco, CA, January 1994.

[11] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

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

[13] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-Safe Disks. In *OSDI '06*, Seattle, WA, November 2006.

[14] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.