Research paper summary:
"De-indirection for Flash-based SSDs with Nameless Writes"

Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau

Federico Wasserman & Rodolphe Lepigre
MOSIG - Parallel, Distributed and Embedded Systems

December 17, 2012

Introduction

Indirection is used in many computer systems. It is used mainly to improve their performance and reliability. For example current hard drives make use of a mapping table that remaps faulty sectors to other locations transparently (from the file system's point of view).

Solid State Drives (SSDs) also use indirection in order to satisfy the specific requirements of their hardware (wear leveling, erase / program cycle, garbage collection). In particular, when a file is modified on an SSD, its original location cannot be updated. The hardware uses copy on write and must make use of indirection to make the filesystem believe that the updated data still resides at the same (virtual) address. The problem is that storing a table to map physical to virtual addresses wastes a lot of expensive flash memory.

In this article, we summarize a paper called *De-Indirection for Flash-Based SSDs with Nameless Writes*, in which Yiyang Zhang et al. propose a way to modify to current SSDs and file systems to remove the need for indirection. Their idea is to let the SSD choose the name (address) of the data to be written and only afterwards inform the file system so that it registers the location in its internal data structures.

Content of the report. In the first section we introduce SSDs and their particular functioning mode. We then present the techniques of indirection that are currently used in SSDs to ensure wear-leveling in the second section. In the third section, Nameless Writes are introduced together with their device interface. The problem of recursive writes is also addressed. Finally, in the fourth section we present an evaluation and show that nameless writes could be a viable solution to solve the indirection issues in SSDs.

SSD principles

Although hard drives appear the same to the file system, independent of their architecture, internally they can be quite different. SSDs do not have the same structures and addressing modes as standard hard drives. They are composed of several flash banks divided into blocks thus the normal cylinder-head-sector addressing is not appropriate, although an indirection is performed to hide this from upper layers.

Because of their inner components, a flash block must be erased before it can be written. As such, data cannot be overwritten in place, so disks use a pattern known as erase-program, requiring not overwritten pages to be moved elsewhere while the block is being erased, to then be put back with the new ones.

A peculiarity of SSDs is that they have a limited number of writes to the same block before they start to show high bit error rates. When this property is mixed with the fact that data needs to be written using an erase-program model, there is a writing multiplication that greatly reduces the SSD lifetime.

To counter these adversities, the firmware inside the SSD handles with indirection where to write data and performs internal garbage collection to recover the best possible block to rewrite, to perform wear-leveling and increase the disk lifetime. Although disks normally hide all their inner workings, after their introduction, file systems started to help them including a command called TRIM. This command informs the disk which pages are not in use anymore so as to aid in the garbage collection mechanism and not reduce the lifespan by much.

Indirection in SSDs

As discussed in the previous section, indirection plays a crucial role in SSDs to increase their lifetime. Indirection is implemented in the Flash Translation Layer (FTL), which maps the virtual to the physical address space.

The main problem with the indirection found in FTLs is that it comes with performance and space overheads.

To better understand the problem, let us consider that we have a 1TB SSD split up into 2KB pages. A full page mapping (with one 32-bit pointer per page) would require a 2GB space, which would be unacceptable considering the cost of SRAM.

Another idea is to map blocks of, say, 128 pages. The space cost of the mapping would be reduced to 32MB which seems more reasonable. However, it has been shown that with such coarse grain mapping, there is a huge performance overhead introduced by garbage collection (Gupta et al.).

A majority of the SSDs that are available to buy use both of these schemes. Most of the data is mapped at block level, but there is a small area that is mapped at the page level for more efficient updates. This technique keeps the space overhead reasonably low while avoiding the high garbage collection cost of the full block-level mapping. The main problem with this technique is that the FTL becomes very complex, and there is still a performance overhead (and it is also harder to maintain). Moreover, garbage collection can still be costly and induce a performance problem.

In all of these designs, the indirection in the FTL implies some cost that might not be very significant now, but when SSDs scale, the hybrid schemes will eventually become infeasible. To solve these problems, the authors of the paper introduce Nameless Writes.

Nameless Writes

The aim of the new technique presented here is to reduce most of the costs of indirection while keeping its benefits. The idea is not to provide any address for a write operation. The device can itself choose where to put the data among all physical pages. The file system is then notified and can record the data in its data-structures. Here is the main interface of a nameless writing device:

```
Nameless_Write(data, len): phys@
Nameless_Overwrite(phys@, data, len): new@
Physical_Read(phys@, len): data
Free(virt/phys@, len)
```

Note that the nameless overwrite has the same mechanism. The original address is given as a parameter, and the new address is returned once the device has made its choice. Also, since a nameless write is an allocating operation, there is a corresponding free operation. In case of a nameless overwrite, the free is handled by the device. Note that for the read operation the physical address is to be provided.

There is but one problem with this Nameless Write interface: the recursive update problem. If the only available write operation is the nameless one, any update to the file system will require a recursive set of updates. For exemple, appending a block at the end of a file will require the inode of the file to be modified using a nameless overwrite which will change its location. So any structure referencing the inode will have to be modified, potentially propagating the update up to the file system's root.

To solve this problem, the authors of the paper propose to provide a segmented address space: a large physical address space for nameless writes, and a small virtual address space. Keeping the pointer-based structures in the virtual space will break the recursion in case of recursive update. The operations provided for virtual read and write are usual:

```
Virtual_Read(virt@, len): data
Virtual_Write(virt@, data, len)
```

Only a very small indirection table is kept by the device to map the virtual addresses to physical addresses, and it will usually fit in the device cache.

Note that physical writes are not allowed, only the device has the power to choose the location of a write. Also, to distinguish a block that is mapped by a virtual block, a flag is set in the block headers.

The device also needs to be able to move data from one place to another in order to allow wear leveling. The interface defines a callback function that can notify the file system of data migration.

```
Migration [Callback] (old_phys@, new@)
```

Blocks in the virtual address space can be moved without having to notice the file system, but for the blocks that are in the physical space, the migration callback has to be used for the file system to update its data structures accordingly.

Evaluation

To evaluate the device interface a series of simulators with different FTLs were created. The first version uses a complete page-level mapping, the second one a hybrid mapping and the third one the nameless-writing structure. The devices try to emulate real life equipment in terms of performance, although they perform somewhat better than todays devices.

The first result presented is the fact that the FTL mapping table size is smallest for Nameless writes when compared to the other two, as a result of having only a small share of the disk that uses this structure. As for the page-level mapping, its size is approximately 10% of the total disk space while the hybrid does not occupy a lot of space, still being bigger than nameless.

In relation to the performance of the different disks, the results show that nameless writing performs similar to page-level mapping, meaning close to the upper-bound as in page-level we have a direct mapping to each page. Both always show results surpassing the ones from a hybrid implementation, especially when the writes are random and the hybrid cannot keep up with the allocation cost.

Finally when considering wear-leveling techniques to reorder data and migrating it to rewrite in unused pages, the results show that there is a small overhead when using Nameless writes instead of pages. This result is expected because of the model, but due to the small overhead and the frequency with which wear-leveling operations are performed, they are not considered dominant or negative.

Conclusion

Nameless Writes have been introduced to reduce the cost of indirection in SSDs. This new interface still ensures wear-leveling and has been tested using an emulated device.

Nameless-writing devices have great advantages, as have been showed with the results of the evaluation. In particular, the space cost of the system is greatly reduced even compared to hybrid mapping FTLs. The design of the FTL is also made much simpler.

Even though the authors only ported the Linux ext3 file system to work with their emulated nameless-writing device, they argue that it would be possible (and will be future work) to do the same with other file systems (ext2, copy-on-write file systems, extent-based file systems).