Improving RAID1 Synchronization Performance Using File System Metadata

Reducing MD RAID1 volume rebuild time.

Hariharan Subramanian
VMware, Inc.
`hari@vmware.com`

Andrei Warkentin
VMware, Inc.
`andreiw@vmware.com`

Alexandre Depoutovitch
VMware, Inc.
`aldep@vmware.com`

Abstract

Linux MD software RAID1 is used ubiquitously by end users, corporations and as a core technology component of other software products and solutions, such as the VMware vSphere® Storage Appliance™ (vSA). MD RAID1 mode provides data persistence and availability in face of hard drive failures by maintaining two or more copies (mirrors) of the same data. vSA makes data available even in the event of a failure of other hardware and software components, e.g. storage adapter, network, or the entire vSphere® server. For recovery from a failure, MD has a mechanism for change tracking and mirror synchronization.

However, data synchronization can consume a significant amount of time and resources. In the worst case scenario, when one of the mirrors has to be replaced with a new one, it may take up to a few days to synchronize the data on a large multi-terabyte disk volume. During this time, the MD RAID1 volume and contained user data are vulnerable to failures and MD operates below optimal performance. Because disk sizes continue to grow at a much faster pace compared to disk speeds, this problem is only going to become worse in the near future.

This paper presents a solution for improving the synchronization of MD RAID1 volumes by leveraging information already tracked by file systems about disk utilization. We describe and compare three different implementations that tap into the file system and assist the MD RAID1 synchronization algorithm to avoid copying unused data. With real-life average disk utilization of 43% [3], we expect that our method will halve the full synchronization time of a typical MD RAID1 volume compared to the existing synchronization mechanism.

1 Introduction

RAID arrays have gained a wide popularity over the last decade. By maintaining data redundancy, they provide a cheap solution for data availability, fault tolerance, and scalability in the event of hardware and software failures [2]. Some of the popular RAID implementations include RAID1, which maintains two or more identical copies of the data over physically separate storage devices, and RAID10 which augments RAID1 with data striping. RAID arrays can be implemented at hardware level, e.g., RAID hardware adapters, as a software product, e.g., Linux MD RAID driver, or as a part of more robust and complex applications, e.g., VMware vSphere® Storage Appliance™ (vSA).

In RAID1, the loss of one copy of the data due to a component failure (e.g., hard drive) is typically followed by an administrative operation, that replaces the failed component with a new one. As part of this, all data needs to be copied (synchronized) to the newly added component. This restores the data redundancy and fault tolerance characteristics. However, storage size has grown exponentially over the recent years, while data access latency and bandwidth improvement rate is significantly smaller. For large arrays, this results in hours during which the array functions below its optimal performance and reliability. Before the synchronization is complete, additional failures may result in data loss and/or unavailability. Therefore, it is very important to minimize synchronization time. In our work, we advocate a new, easy to implement method that reduces the amount of data that needs to be synchronized and consequently decreases the synchronization time.

We implemented our method in the Linux MD software RAID1 driver and integrated it with the VMware vSA product. The key observation behind our method

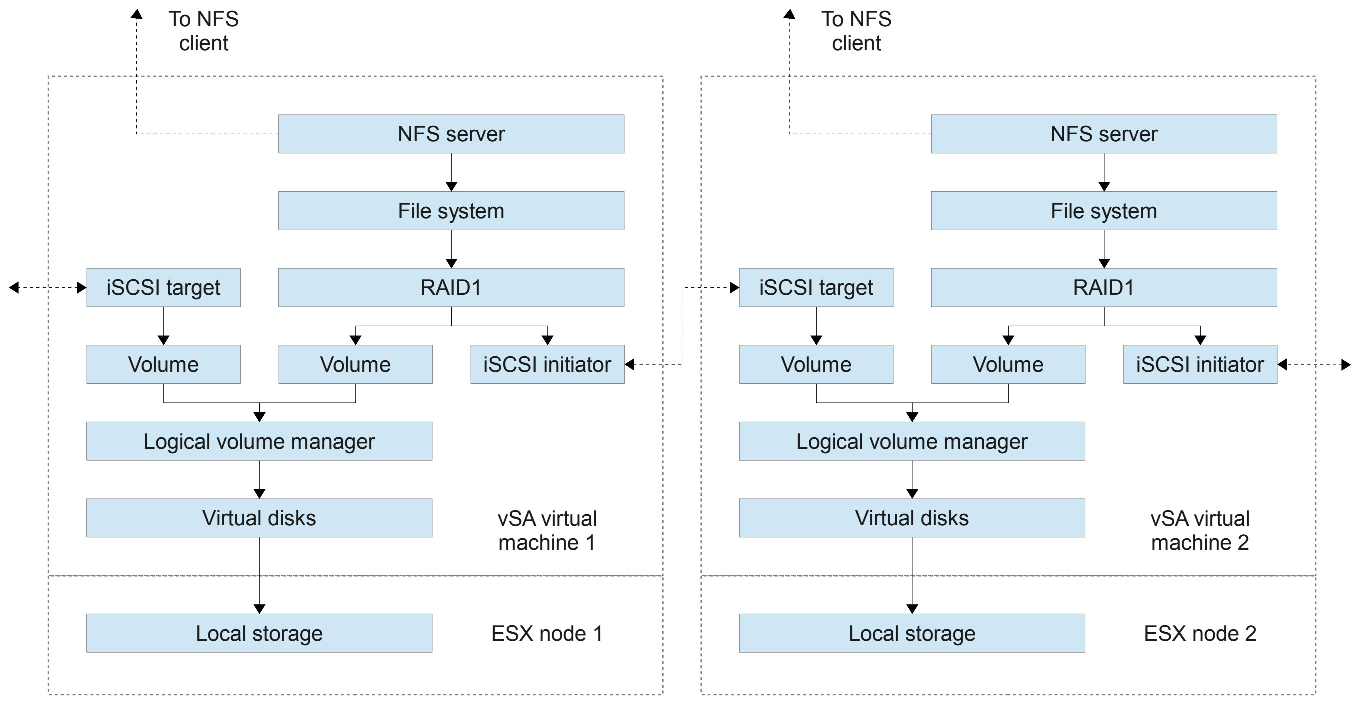

Figure 1: vSA architecture overview.

is that, since the synchronization is in the block device layer, the MD RAID1 mirroring driver needlessly synchronizes unallocated file system blocks, in addition to blocks containing useful data. We investigated three different approaches of transferring the unused block list from the file system to the synchronization algorithm. All three approaches populate the unused block list and change synchronization logic to take a list of unallocated blocks into account. They require minimal changes to the existing MD RAID1 control flow.

The approaches differ in how the unused data blocks are obtained. The first approach leverages user space file system utilities (specifically, `e2fsprogs` in an EXT4 file system) to obtain a list of unused file system blocks. A user space helper application uses the list to construct a bitmap representing the blocks that are currently in use by the file system. It then passes the bitmap to the MD RAID1 kernel driver, where it is used to skip copying the disk blocks that are not in use by the file system. The second approach continuously tracks the blocks not in use by the file system by intercepting discard requests to the block device (called `REQ_DISCARD` in the Linux kernel). The final, hybrid approach, avoids the overhead of maintaining in-memory unused block state of the previous approach, while also taking advantage of a user space helper, albeit in a way which is much simpler and independent of file-system implementation. It utilizes the Linux kernel `FITRIM` `ioctl` to force the file system to send `REQ_DISCARD` requests for unused blocks.

2 vSphereStorage Appliance

Linux MD RAID is not only widely used by the end users, it is also an important building block for larger and more complex software products. One such example is the vSphere® Storage Appliance™ (vSA) developed by VMware and released as a commercial product last year [1]. This software provides shared storage benefits such as reliability, availability, and shared access without needing to buy complex and expensive specialized shared storage hardware. The high level architecture of the vSA storage stack is presented at Figure 1. vSA consists of two or more hardware nodes with ESX server installed, running a virtual machine with Linux and vSA software.

The vSA software exports data to clients through the NFSv3 protocol. EXT4 file system is used to store the user data. In order to provide reliability and availability, the Linux MD RAID driver is utilized to duplicate data between the hardware nodes. Access to the remote node storage is done through the iSCSI protocol over the network. Data synchronization speed between nodes is often limited by the 1Gbit network link. Because vSA targets the small and medium business (SMB) market, cost savings is an important criteria. Therefore, an upgrade to a 10Gbit network is often undesirable due to the high cost involved in replacing not only network cards but routers, switches, and other infrastructure components.

In this environment, optimization of data transfers is very important. Without our mechanism, it takes approximately 5 hours to synchronize a typical 2TiB vSA data volume after one of the nodes was replaced. During this time, the vSA functions in degraded mode with decreased performance and without fault tolerance. With our mechanism in place and an average storage utilization of 43% [3], this critical time is reduced to slightly more than 2 hours. We believe that not only vSA, but other projects and products involving storage replication would benefit from our mechanism.

3 Control flow

Our method introduces changes to the MD RAID1 array algorithm only during array synchronization. Therefore, there is no additional run-time overhead for any array functions in regular or degraded modes compared to vanilla Linux MD driver. Our mechanism comes into play whenever RAID synchronization is triggered automatically or upon user request.

Upon receiving such request, depending on whether a full or incremental synchronization is required, the vanilla Linux MD driver copies either all blocks to the new block device, or only those blocks marked in the internal write-intent MD bitmap as changed since the time the array was healthy and fully synchronized. If a write request arrives while synchronization is in progress, the MD RAID driver pauses the synchronization process until the write operation is complete on all disks. This prevents race conditions between synchronization and regular writes.

In all of our approaches, any IO error encountered by the RAID block devices used by MD is reported to a user mode helper program. In case of the vSA, this is a Java-based application referred to as the vSA business logic software. This helper application is responsible for handling detected IO errors, detecting that a previous hardware failure has been rectified and re-introducing previously failed block devices back into a degraded MD RAID volume. Thereby, this helper program controls the point at which MD synchronization process starts.

In our mechanism, upon receiving a synchronization request, the MD driver starts synchronizing blocks marked in the write-intent bitmap to all degraded devices just as described above, but it queries an additional in-use/unused block list. Depending on the actual implementation, this list might come from a user-space helper program or from handling an `FITRIM ioctl` issued to the file system layered on top of the RAID volume. With the list of unused blocks, the MD driver can proceed to synchronize only blocks that are both marked as in-use and marked in the write-intent bitmap.

While synchronization is in progress, previously unused blocks may become allocated again, but this is equivalent to the concurrent writes and synchronization case above. In this case, before any attempt to read such a block, a write must be issued to initialize the block data. This write will always be propagated to all devices during synchronization, as previously described. If no write was issued to a previously unused block before reading, an application cannot depend on the read contents of such a block, thus there should be no need to synchronize it.

There is one important difference in the behavior of the vanilla Linux MD driver and our modified MD driver. If a block has been reported as unused to our synchronization mechanism, is subsequently allocated and read from several times without being written to, the vanilla Linux MD driver would return the same data on every read. With our mechanism in place, different reads to uninitialized data might return different data depending on the device the read operation was dispatched to. However, all POSIX-compliant file system returns zeroes for reads to unwritten parts of a file. We can not think of any correctly written code that depends on such behavior, but if such software exist, it should not be used with our mechanism.

4 Metadata snapshot

The first approach introduces no run-time overhead to the I/O and data synchronization paths of the RAID driver, and is only involved after a hardware failure is detected and rectified. The approach involves the following steps,

- Obtaining list of unused blocks from the file system.

- Representing the data in an in-use bitmap.
- Injecting the in-use bitmap to MD using a new `ioctl` call.

The list of unused blocks is obtained out-of-band by examining on-disk file system structures. For vSA, this involves examining the EXT4 file system on the MD RAID device. `dumpe2fs`, a common system utility from the `e2fsprogs` package, is used to query file system metadata and statistics for the EXT2, EXT3 and EXT4 file systems. The vSA business logic software, which controls disk creation, failure detection and other management operations, parses `dumpe2fs` output to generate a list of unused blocks from the vSA file system. The list contains ranges of blocks in units of file system block size. The information is presented for each file system block group as a comma separated list of unused block ranges.

The control flow for performing MD re-synchronization is presented at Figure 2.

The data obtained is used to populate a bitmap representing blocks that are currently used by the file system. Instead of extending the existing in-memory write intent state, a separate bitmap was used. The write-intent bitmap divides the disk into chunks, and keeps track of which disk chunks have modifications that need to be synchronized to disks that are currently unavailable. A separate bitmap enables us to pick a different granularity for tracking used blocks, with the intent of investigating optimum granularity for tracking such information. This flexibility is potentially worth the additional complexity and memory overhead of maintaining a new bitmap.

The in-use bitmap divides the MD device into equal sized chunks. A chunk is always larger than the size of a file system block, and would ideally match the granularity of an individual synchronization I/O. The bitmap comprises a series of pages, each covering 32768 chunks. With the additional in-use bitmap, the modified MD RAID1 synchronization algorithm determines if the synchronization I/O being performed can be skipped. The actual changes to the main routine involved are minimal.

The in-use bitmap is injected into the MD RAID1 driver using a new `ioctl` mechanism. The MD RAID1 driver is modified to accept a bitmap solely while synchronization operation is ongoing. Once the synchronization operation is complete, the bitmap is automatically cleared.

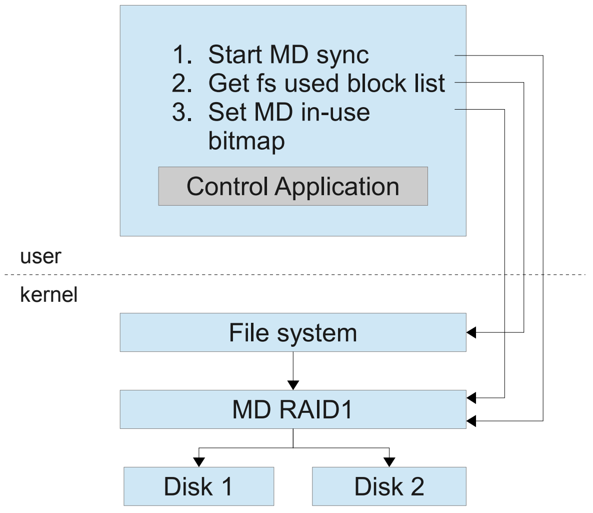

Figure 2: Control flow for MD resync with the metadata snapshot approach.

This avoids any data consistency issues resulting from possible malicious use of the interface, and follows good security practices. Since the vSA business logic software completely controls when an MD synchronization operation occurs, it would not be possible for an out-of-date bitmap to be applied, avoiding possible data corruption issues.

5 Discard request tracking

The second approach of exposing file system unused block information to the MD driver relies on an already existing set of functionality present in the Linux kernel. The Linux block I/O subsystem provides a way to notify hardware that a range of blocks is not in use anymore by upper layers. A file system might thus send `REQ_DISCARD` requests when a file is deleted. The original aim of this functionality was to enable more intelligent wear-leveling mechanisms for solid-state storage, yet it is used in implementing thin-provisioned SCSI LUNs and provides the data we need to avoid synchronizing unused blocks. The method by which unused block ranges are sent into the block layer is a `REQ_DISCARD` I/O request. Just like any other I/O operation, it consists of a start block and the number of blocks affected, and arrives at the same MD I/O dispatch routine handling regular accesses. This implies that the MD driver has to keep track of blocks being marked as used and unused. A block is marked as being unused when a `REQ_DISCARD` request for it arrives, and is marked as being in use on a write request. In practice, there are real-life restrictions that limit the usefulness of an approach based purely on live tracking of `REQ_DISCARD` requests, as we shall see.

The first idea that comes to mind is to track the in-use/unused state in the same memory and disk structures already used to keep track of write intent state. Using the same memory structures has the implication of no extra memory overhead[1], and the bit twiddling is done at the same code location and under the same locks, meaning that the run-time overhead is the least of other possible approaches. The new state is persisted across reboots by extending the on-disk write intent bitmap with an in-use/unused bit. Of course, the RAID1 I/O dispatch routine also needs to be changed to handle `REQ_DISCARD` block I/O by marking the affected blocks as unused and finishing the I/O.

Unfortunately, in a real life setting, where each bitmap chunk is significantly larger than the typical I/O size[2], this approach gives poor results. A typical discard I/O request acts on a range of kibibytes, usually with a granularity of 4KiB or so, and such requests are basically lost without keeping track of discard requests at a finer granularity than the bitmap chunk. We could maintain a separate bitmap, say at 4KiB granularity, but the memory overhead of such a bitmap are enormous at typical capacities[3]. The disk persistence is an additional problem. Extending the write-intent bitmap carries the same granularity issues present with reusing the in-memory state, while maintaining a separate more granular bitmap would result in additional disk I/Os, lowering the write performance. Additionally, the change in internal RAID metadata brings upgradeability implications where the existing structures need to be replaced by larger ones.

Support of upgrade from an earlier version of MD metadata to a newer one aware of the in-use/unused blocks, means we need to have some method of registering blocks not in use by the file system with MD. Fortunately, there is no need for a new interface to achieve this. The `FITRIM` file system `ioctl` causes the file system to send `REQ_DISCARD` I/O requests for all unused blocks. At the current moment, file systems do not persist in-use/unused knowledge across remounts, thus `FITRIM` is a sufficient method to initialize our MD in-use/unused state. Given the issues with on-disk persistence, we might as well rely on `FITRIM` to initialize our in-memory state after mounting the file system.

To address the issues surrounding tracking `REQ_DISCARD` requests, we can switch from a bitmap to a data structure that makes it more convenient to store unused block ranges. Such a structure is a special interval tree that coalesces overlapping and sequential ranges, implemented using a red-black tree. The time complexity is $O(\log n)$, especially if you assume the total number of intervals to be pretty low. The changes to MD to enable the use of discard ranges as an optimization are minimal, and the synchronization overhead can be mitigated by employing a relativistic red-black tree algorithm [4] instead of the default Linux rb-tree implementation. The Achilles heel of this approach, however, is the worst-case memory overhead. An access pattern of small discards, such as interleaved 4KiB accesses, will result in an overhead 32 times worse than the equivalent overhead of using a bitmap with 4KiB granularity. Some of this can be mitigated by enforcing a minimum granularity and pruning the range tree based on memory pressure, but all of this added complexity basically nullifies the original advantages of storing ranges.

6 Hybrid Approach

The previous approach to tracking `REQ_DISCARD` requests relies on the assumption that it is typical to expect discard requests over the normal lifetime of an on-line RAID array with a mounted file system. However, that is not the case. While a file system like EXT4 certainly could be mounted in a way that will generate discard requests for every file erase, that was generally avoided in older kernels due to discard requests being processed synchronously and acting as barriers, impacting I/O performance. If we consider that we always issue `FITRIM` during RAID synchronization, then we just have to handle `REQ_DISCARD` requests during synchronization time. At synchronization time we can employ simple logic to track consecutive discards, marking the affected chunks in the separate bitmap as not in use. This is effectively the combination of the first approach with first idea considered in the previous section, without the on-disk persistence changes and with a more intelligent and restrictive algorithm for marking chunks as being not in use.

[1]We end up stealing a bit from the per bitmap-chunk field to describe the new in-use/unused state, which has has the largely irrelevant implication of reducing the number of concurrent I/O per block chunk from 16383 to 8191.

[2]For a 100MiB disk, the default bitmap chunk is 4KiB, while for a 10TiB disk, the bitmap chunk size is usually 64MiB.

[3]A 10TiB disk would need a 320MiB bitmap.

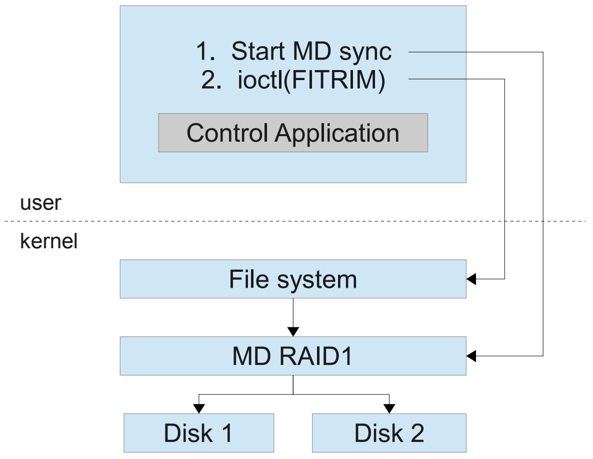

Figure 3: Control flow for MD resync with the hybrid approach.

It is better than the previous approach to tracking discards because it covers the case of out-of-order small discards that ultimately add up to a larger unused block—we let the file system handle coalescing and ordering these at `FITRIM` time. Using a separate bitmap, with a granularity smaller than the write intent bitmap, improves the case where the write intent chunk size is too large to effectively use this algorithm.

The control flow for performing MD re-synchronization is presented at Figure 3.

7 Discussion

Each of the above approaches to obtain the list of unused blocks has its own advantages and disadvantages. The main advantage of using a file system metadata is its independence on the Linux kernel version deployed on the system. Certain distributions, such as all versions of Red Hat Enterprise Linux or SUSE Enterprise Linux, except the latest SLES11SP2, do not support `REQ_DISCARD` or `FITRIM` functionality. This makes usage of a user space helper the only way to get the list of unused blocks, given that back porting changes to the kernel block I/O subsystem and file systems is no trivial matter. This approach does not depend on kernel version, which makes it applicable to most of currently deployed systems. Another advantage is that more functionality remains in user space, which makes the code more reliable and easier to debug. The disadvantage of this approach is its dependence on user mode utilities which are not standardized, which are file system-dependent, and which may change their output format in future versions.

The advantage of discard request tracking is in the transparency of the approach. There is no need for anything special to occur to make use of this functionality to improve synchronization, other than ensuring that the file system generates discard requests for erased files. The disadvantage of live tracking is in its memory consumption. The bitmap based approach, with a sufficiently small granularity to achieve effectiveness, would consume an additional 300MiB of kernel memory. A range-based approach would scale the memory usage, but become effectively unbounded with severe file system fragmentation. Therefore, we would prefer the hybrid approach.

The advantage of obtaining the list of unused blocks through `FITRIM` `ioctl` command is in the use of a standard interface, recommended for use by all general purpose file systems. In current kernels, the `FITRIM` interface supported by EXT3, EXT4, btrfs, xfs, ocfs, and others. We would, however, like to see better defined documentation on `FITRIM` behavior. For example, to the best of our knowledge, the ordering of blocks to be discarded is not described, nor are any guarantees regarding the blocks reported for file systems that persist discarded block information. This allows some file system implementations to report unused blocks out-of-order, or to only report changes since the last `FITRIM` even across remounts, both of which will negatively affect the hybrid approach.

8 Conclusion

In this paper, we proposed a new method for RAID array synchronization. This method requires minimal changes to the existing Linux kernel code. It reduces synchronization time by a factor of two in the common case, thus improving reliability and performance of the RAID array.

We investigated and compared different implementation methods and highlighted their strong and weak points. The optimal granularity of the unused block bitmap needs to be further investigated, and we are planning to extend the synchronization improvements to other RAID levels provided by the MD driver. Furthermore, the `FITRIM` kernel interface needs to be better defined to address the ordering and persistence concerns noted in the discussion section above.

9 Acknowledgments

Many thanks to Edward Goggin and Aju John for the peer review and support of this research.

References

[1] VMWare vSphere Storage Appliance Technical Deep Dive, 2011.

[2] Conference on File and Storage Technologies. *RAID: high-performance, reliable secondary storage*, 1994.

[3] Conference on File and Storage Technologies. *A study of practical deduplication*, 2011.

[4] P. W. Howard and J. Walpole. Relativistic red-black trees. December 2010.