dsync: Efficient Block-wise Synchronization of Multi-Gigabyte Binary Data

Thomas Knauth
Technische Universität Dresden

Christof Fetzer
Technische Universität Dresden

Abstract

Backing up important data is an essential task for system administrators to protect against all kinds of failures. However, traditional tools like `rsync` exhibit poor performance in the face of today's typical data sizes of hundreds of gigabytes. We address the problem of *efficient*, periodic, multi-gigabyte state synchronization. In contrast to approaches like `rsync` which determine changes after the fact, our approach tracks modifications online. Tracking obviates the need for expensive checksum computations to determine changes. We track modification at the block-level which allows us to implement a very efficient delta-synchronization scheme. The block-level modification tracking is implemented as an extension to a recent 3.2 Linux kernel.

With our approach, named dsync, we can improve upon existing systems in several key aspects: disk I/O, cache pollution, and CPU utilization. Compared to traditional checksum-based synchronization methods dsync decreases synchronization time by up to two orders of magnitude. Benchmarks with synthetic and real-world workloads demonstrate the effectiveness of dsync.

1 Introduction

"Everything fails all the time." is the modus operandi when it comes to provisioning critical IT infrastructure. Although the definition of critical is up for debate, redundancy is key to achieving high availability. Redundancy can be added at many levels. For example, at the hardware level, deploying two network cards per server can allow one network card to fail, yet the server will still be reachable. Performance may be degraded due to the failure but the server is still available.

Adding hardware redundancy is just one piece of the availability puzzle. To ensure data availability in the presence of failures, the data must be replicated. However, synchronizing tens, hundreds, or even thousands of gigabyte of data across the network is expensive. It is expensive in terms of network bandwidth, if a naïve copy-everything approach is used. It is also expensive in terms of CPU cycles, if a checksum-based delta-copy approach is used. Although a delta-copy minimizes network traffic, it relies on a mechanism to identify differences between two versions of the data in question. Determining the differences after the fact is less efficient than recording modifications while they are happening.

One problem with synchronizing large amounts of data, e.g., for backups, is that the backup operation takes on the order of minutes to hours. As data sets continue to grow, consumer drives now hold up to 4 TB, so does the time required to synchronize them. For example, just reading 4 TB stored on a single spinning disk takes more than 6 hours [14]. Copying hundreds of gigabytes over a typical wide area network for remote backups will proceed at a fraction of the previously assumed 170 MB/s. Ideally, the time to synchronize should be independent of the data size; with the size of updated/added data being the main factor influencing synchronization speed.

The key insight is, that between two synchronizations of a data set, most of the data is unmodified. Hence, it is wasteful to copy the entire data set. Even if the data sets are almost identical, the differences have to be determined. This is done, for example, by computing block-wise checksums. Only blocks with mismatching checksums are transferred. Instead of detecting changes after the fact, we propose to track and record them at run time. Online tracking obviates checksum computations, while still only transferring the changed parts of the data. The benefits of online modification recording are plentiful: (1) minimizes network traffic, (2) no CPU cycles spent on checksum computation, (3) minimizes the data read from and written to disk, and (4) minimizes page cache pollution.

We implemented a prototype of our synchronization solution, named dsync, on Linux. It consists of a kernel modification and two complimentary userspace

tools. The kernel extension tracks modifications at the block device level. The userspace tools, `dmextract` and `dmmerge`, allow for easy extraction and merging of the modified block level state.

To summarize, in this paper, we make the following contributions:

- Identify the need for better mechanisms to synchronize large, binary data blobs across the network.
- Propose an extension to the Linux kernel to enable efficient synchronization of large, binary data blobs.
- Extend the Linux kernel with block-level tracking code.
- Provide empirical evidence to show the effectiveness of our improved synchronization method.
- We share with the scientific community all code, measurements, and related artifacts. We encourage other researchers to replicate and improve our findings. The results are available at `https://bitbucket.org/tknauth/devicemapper/`.

2 Problem

The task at hand is to periodically synchronize two physically distant data sets and to do so efficiently. The qualifier "periodically" is important because there is little to optimize for a one-off synchronization. With periodic synchronizations, on the other hand, we can potentially exploit the typical case where the majority of the data is unmodified between successive synchronizations.

There exists no domain-specific knowledge about the data being synchronized, i.e., we have to treat it as a binary blob. Using domain-specific knowledge, such as file system meta-data, alternative optimizations are possible. Synchronization tools routinely use a file's last modified time to check whether to consider it for a possible transfer.

We are primarily interested in large data sizes of multiple giga- to terabytes. The techniques we present are also applicable to smaller sizes, but the problems we solve with our system are more pressing when data sets are large. One example in the cloud computing environment are virtual machine disks. Virtual machine disks change as a result of the customer starting a virtual machine, performing computation, storing the result, and shutting the virtual machine down again. As a result of the users' actions, the disk's contents change over time. However, only a fraction of the entire disk is actually modified. It is the cloud provider's responsibility to store the virtual machine disk in multiple locations, e.g., for fault tolerance. If one data center becomes unavailable, the customer can restart their virtual machine in a backup data center. For example, a cloud provider may synchronize virtual machine disks once per day between two data centers A and B. If data center A becomes unavailable, data center B has a copy which is at most 24 hours out of date. If customers need more stringent freshness guarantees, the provider may offer alternative backup solutions to the ones considered in this paper.

Copying the entire data is a simple and effective way to achieve synchronization. Yet, it generates a lot of gratuitous network traffic, which is unacceptable. Assuming an unshared 10 Gigabit Ethernet connection, transferring 100 GB takes about 83 seconds (in theory anyway and assuming an ideal throughput of 1.2 GB/s). However, 10 Gigabit Ethernet equipment is still much more expensive than commodity Gigabit Ethernet. While 10 Gigabit may be deployed inside the data center, wide-area networks with 10 Gigabit are even rarer. Also, network links will be shared among multiple participants – be they data streams of the same applications, different users, or even institutions.

The problem of transmitting large volumes of data over constrained long distance links, is exacerbated by continuously growing data sets and disk sizes. Offsite backups are important to provide disaster recovery and business continuity in case of site failures.

Instead of indiscriminatly copying everything, we need to identify the changed parts. Only the changed parts must actually be transmitted over the network. Tools, such as `rsync`, follow this approach. The idea is to compute one checksum for each block of data at the source and destination. Only if there is a checksum mismatch for a block, is the block transferred. While this works well for small data sizes, the checksum computation is expensive if data sizes reach into the gigabyte range.

As pointed out earlier, reading multiple gigabytes from disks takes on the order of minutes. Disk I/O operations and bandwidth are occupied by the synchronization process and unavailable to production workloads. Second, checksum computation is CPU-intensive. For the duration of the synchronization, one entire CPU is dedicated to computing checksums, and unavailable to the production workload. Third, reading all that data from disk interferes with the system's page cache. The working set of running processes is evicted from memory, only to make place for data which is used exactly *once*. Applications can give hints to the operating system to optimize the caching behavior [3]. However, this is not a perfect solution either, as the OS is free to ignore the advice if it cannot adhere to it. In addition, the application developer must be aware of the problem to incorporate hints into the program.

All this is necessary because there currently is no way of identifying changed blocks without comparing their

checksums. Our proposed solution, which tracks block modifications as they happen, extends the Linux kernel to do just that. The implementation details and design considerations form the next section.

3 Implementation

A block device is a well known abstraction in the Linux kernel. It provides random-access semantics to a linear array of blocks. A block typically has the same size as a page, e.g., 4 KiB (2^{12} bytes). The actual media underlying the block device may consist of even smaller units called sectors. However, sectors are not addressable by themselves. Sectors are typically 512 byte in size. Hence, 8 sectors make up a block. Block devices are generally partitioned and formatted with a file system. For more elaborate use cases, the Linux device mapper offers more flexibility to set up block devices.

3.1 Device mapper

The Linux device mapper is a powerful tool. The device mapper, for example, allows multiple individual block devices to be aggregated into a single logical device. In device mapper terminology, this is called a linear mapping. In fact, logical devices can be constructed from arbitrary contiguous regions of existing block devices. Besides simple aggregation, the device mapper also supports RAID configurations 0 (striping, no parity), 1 (mirroring), 5 (striping with distributed parity), and 10 (mirroring and striping). Another feature, which superficially looks like it solves our problem at hand, is snapshots. Block devices can be frozen in time. All modifications to the original device are re-directed to a special copy-on-write (COW) device. Snapshots leave the original data untouched. This allows, for example, to create consistent backups of a block device while still being able to service write requests for the same device. If desired, the external modifications can later be merged into the original block device. By applying the external modifications to a second (potentially remote) copy of the original device, this would solve our problem with zero implementation effort.

However, the solution lacks in two aspects. First, additional storage is required to temporarily buffer all modifications. The additional storage grows linearly with the number of modified blocks. If, because of bad planning, the copy-on-write device is too small to hold all modifications, the writes will be lost. This is unnecessary to achieve what we are aiming for. Second, because modifications are stored out-of-place, they must also be merged into the original data at the source of the actual copy; in addition to the destination. Due to these limitations we consider device mapper snapshots as an inappropriate solution to our problem.

Because of the way the device mapper handles and interfaces with block devices, our block-level tracking solution is built as an extension to it. The next section describes how we integrated the tracking functionality into the device mapper,

3.2 A Device Mapper Target

The device mapper's functionality is split into separate targets. Various targets implementing, for example, RAID level 0, 1, and 5, already exist in the Linux kernel. Each target implements a predefined interface laid out in the `target_type` [1] structure. The `target_type` structure is simply a collection of function pointers. The target-independent part of the device mapper calls the target-dependant code through one of the pointers. The most important functions are the constructor (`ctr`), destructor (`dtr`), and mapping (`map`) functions. The constructor is called whenever a device of a particular target type is created. Conversely, the destructor cleans up when a device is dismantled. The userspace program to perform device mapper actions is called `dmsetup`. Through a series of `ioctl()` calls, information relevant to setup, tear down, and device management is exchanged between user and kernel space. For example,

```
# echo 0 1048576 linear /dev/original 0 | \
  dmsetup create mydev
```

creates a new device called mydev. Access to the sectors 0 through 1048576 of the mydev device are mapped to the same sectors of the underlying device `/dev/original`. The previously mentioned function, `map`, is invoked for every access to the linearly mapped device. It applies the offset specified in the mapping. The offset in our example is 0, effectively turning the mapping into an identity function.

The device mapper has convenient access to all the information we need to track block modifications. Every access to a mapped device passes through the `map` function. We adapt the `map` function of the linear mapping mode for our purposes.

3.3 Architecture

Figure 1 shows a conceptual view of the layered architecture. In this example we assume that the tracked block device forms the backing store of a virtual machine (VM). The lowest layer is the physical block device, for example, a hard disk. The device mapper can be used

[1] `http://lxr.linux.no/linux+v3.6.2/include/linux/device-mapper.h#L130`

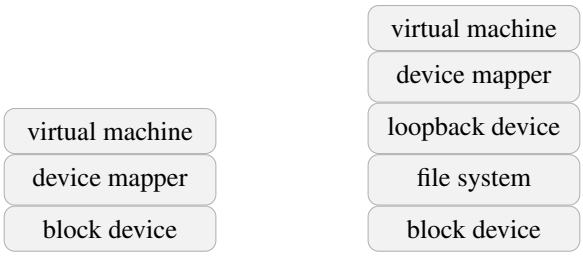


Figure 1: Two configurations where the tracked block device is used by a virtual machine (VM). If the VM used a file of the host system as its backing store, the loopback device turns this file into a block device (right).

to create a tracked device directly on top of the physical block device (Figure 1, left). The tracked block device replaces the physical device as the VM's backing store.

Often, the backing store of a virtual machine is a file in the host's filesystem. In these cases, a loopback device is used to convert the file into a block device. Instead of tracking modifications to a physical device, we track modifications to the loopback device (Figure 1, right). The tracked device again functions as the VM's backing store. The tracking functionality is entirely implemented in the host system kernel, i.e., the guests are unaware of the tracking functionality. The guest OS does not need to be modified, and the tracking works with all guest operating systems.

3.4 Data structure

Storing the modification status for a block requires exactly one bit: a set bit denotes modified blocks, unmodified blocks are represented by an unset bit. The status bits of all blocks form a straightforward bit vector. The bit vector is indexed by the block number. Given the size of today's hard disks and the option to attach multiple disks to a single machine, the bit vector may occupy multiple megabytes of memory. With 4 KiB blocks, for example, a bit vector of 128 MiB is required to track the per-block modifications of a 4 TiB disk. An overview of the relationship between disk and bit vector size is provided in Table 1.

The total size of the data structure is not the only concern when allocating memory inside the kernel; the size of a single allocation is also constrained. The kernel offers three different mechanisms to allocate memory: (1) `kmalloc()`, (2) `__get_free_pages()`, and (3) `vmalloc()`. However, only `vmalloc()` allows us to reliably allocate multiple megabytes of memory with a single invocation. The various ways of allocating Linux kernel memory are detailed in "Linux Device Drivers" [7].

Total memory consumption of the tracking data structures may still be a concern: even commodity (consumer) machines commonly provide up to 5 SATA ports for attaching disks. Hard disk sizes of 4 TB are standard these days too. To put this in context, the block-wise dirty status for a 10 TiB setup requires 320 MiB of memory. We see two immediate ways to reduce the memory overhead:

1. Increase the minimum unit size from a single block to 2, 4, or even more blocks.

2. Replace the bit vector by a different data structure, e.g., a bloom filter.

A bloom filter could be configured to work with a fraction of the bit vector's size. The trade-off is potential false positives and a higher (though constant) computational overhead when querying/changing the dirty status. We leave the evaluation of tradeoffs introduced by bloom filters for future work.

Our prototype currently does not persist the modification status across reboots. Also, the in-memory state is lost, if the server suddenly loses power. One possible solution is to persist the state as part of the server's regular shutdown routine. During startup, the system initializes the tracking bit vector with the state written at shutdown. If the initialization state is corrupt or not existing, each block is marked "dirty" to force a full synchronization.

3.5 User-space interface

The kernel extensions export the relevant information to user space. For each device registered with our customized device mapper, there is a corresponding file in `/proc`, e.g., `/proc/mydev`. Reading the file gives a human-readable list of block numbers which have been written. Writing to the file resets the information, i.e., it clears the underlying bit vector. The `/proc` file system integration uses the `seq_file` interface [15].

Extracting the modified blocks from a block device is aided by a command line tool called `dmextract`. The `dmextract` tool takes as its only parameter the name of the device on which to operate, e.g., `# dmextract mydevice`. By convention, the block numbers for `mydevice` are read from `/proc/mydevice` and the block device is found at `/dev/mapper/mydevice`. The tool outputs, via standard out, a sequence of $(blocknumber, data)$ pairs. Output can be redirected to a file, for later access, or directly streamed over the network to the backup location. The complementing tool for block integration, `dmmerge`, reads a stream of information as produced by `dmextract` from standard input, A single parameter points to the block device into which the changed blocks shall be integrated.

Disk size	Disk size (bytes)	Bit vector size (bits)	Bit vector size (bytes)	Bit vector size (pages)	Bit vector size
4 KiB	2^{12}	2^{0}	2^{0}	2^{0}	1 bit
128 MiB	2^{27}	2^{15}	2^{12}	2^{0}	4 KiB
1 GiB	2^{30}	2^{18}	2^{15}	2^{3}	64 KiB
512 GiB	2^{39}	2^{27}	2^{24}	2^{12}	16 MiB
1 TiB	2^{40}	2^{28}	2^{25}	2^{13}	32 MiB
4 TiB	2^{42}	2^{30}	2^{27}	2^{15}	128 MiB

Table 1: Relationship between data size and bit vector size. The accounting granularity is 4 KiB, i.e., a single block or page.

Following the Unix philosophy of chaining together multiple programs which each serve a single purpose well, a command line to perform a remote backup may look like the following:

```
# dmextract mydev | \
  ssh remotehost dmmerge /dev/mapper/mydev
```

This extracts the modifications from `mydev` on the local host, copies the information over a secure channel to a remote host, and merges the information on the remote host into an identically named device.

4 Evaluation

The evaluation concentrates on the question of how much the synchronization time decreases by knowing the modified blocks in advance. We compare dsync with four other synchronization methods: (a) `copy`, (b) `rsync`, (c) `blockmd5sync`, (d) ZFS send/receive. `blockmd5sync` is our custom implementation of a lightweight `rsync`. The following sections cover each tool/method in more detail.

4.1 Synchronization tools

4.1.1 scp/nc

`scp`, short for secure copy, copies entire files or directories over the network. The byte stream is encrypted, hence *secure* copy, putting additional load on the endpoint CPUs of the transfer. Compression is optional and disabled for our evaluation. The maximum throughput over a single encrypted stream we achieved with our benchmark systems was 55 MB/s using the (default) aes128-ctr cipher. This is half of the maximum throughput of a 1 gigabit Ethernet adapter. The achievable network throughput for our evaluation is CPU-bound by the single threaded SSH implementation. With a patched version of ssh [2] encryption can be parallelized for some ciphers, e.g., aes128-ctr. The application level throughput of this parallelized version varies between 70 to 90 MB/s. Switching to a different cipher, for example, aes128-cbc, gives an average throughput of 100 MB/s.

To transfer data unencrypted, `nc`, short for netcat, can be used as an alternative to ssh. Instead of netcat, the patched version of `ssh` also supports unencrypted data transfers. Encryption is dropped after the initial secure handshake, giving us a clearer picture of the CPU requirements for each workload. The throughput for an unencrypted ssh transfer was 100 MB/s on our benchmark systems. We note, however, that whilst useful for evaluation purposes, disabling encryption in a production environment is unlikely to be acceptable and has other practical disadvantages, for example, encryption also helps to detect (non-malicious) in-flight data corruption.

4.1.2 rsync

`rsync` is used to synchronize two directory trees. The source and destination can be remote in which case data is transferred over the network. Network transfers are encrypted by default because `rsync` utilizes secure shell (`ssh`) access to establish a connection between the source and destination. If encryption is undesirable, the secure shell can be replaced by its unencrypted sibling, `rsh`, although we again note that this is unlikely to be acceptable for production usage. Instead of `rsh`, we configured `rsync` to use the drop-in ssh replacement which supports unencrypted transfers. `rsync` is smarter than `scp` because it employs several heuristics to minimize the transferred data. For example, two files are assumed unchanged if their modification time stamp and size match. For potentially updated files, `rsync` computes block-wise checksums at the source and destination. In addition to block-wise checksums, the sender computes rolling checksums. This allows `rsync` to efficiently handle shifted content, e.g., a line deleted from a configuration file. However, for binary files this creates a huge computational overhead. Only if the checksums for a block are different is that block transferred to the

[2] `http://www.psc.edu/index.php/hpn-ssh`

destination. For an in-depth description of the algorithm please refer to the work of Tridgell and Mackerras [17]. While `rsync` minimizes the amount of data sent over the network, computing checksums for large files poses a high CPU overhead. Also note, that the checksum computation takes place at both the source *and* destination, although only the source computes rolling checksums.

4.1.3 Blockwise checksums (`blockmd5sync`)

`rsync`'s performance is limited by its use of rolling checksums at the sender. If we discard the requirement to detect shifted content, a much simpler and faster approach to checksum-based synchronization becomes feasible. We compute checksums only for non-overlapping 4KiB blocks at the sender and receiver. If the checksums for block B_i do not match, this block is transferred. For an input size of N bytes, $\lceil N/B \rceil$ checksums are computed at the source and target, where B is the block size, e.g., 4 kilobytes. The functionality is implemented as a mix of Python and bash scripts, interleaving the checksum computation with the transmission of updated blocks. We do not claim our implementation is the most efficient, but the performance advantages over `rsync` will become apparent in the evaluation section.

4.1.4 ZFS

The file system ZFS was originally developed by Sun for their Solaris operating system. It combines many advanced features, such as logical volume management, which are commonly handled by different tools in a traditional Linux environment. Among these advanced features is snapshot support; along with extracting the difference between two snapshots. We include ZFS in our comparison because it offers the same functionality as dsync albeit implemented at a different abstraction level. Working at the file system layer allows access to information unavailable at the block level. For example, updates to paths for temporary data, such as `/tmp`, may be ignored during synchronization. On the other hand, advanced file systems, e.g., like ZFS, may not be available on the target platform and dsync may be a viable alternative. As ZFS relies on a copy-on-write mechanism to track changes between snapshots, the resulting disk space overhead must also be considered.

Because of its appealing features, ZFS has been ported to systems other than Solaris ([1], [9]. We use version 0.6.1 of the ZFS port available from `http://zfsonlinux.org` packaged for Ubuntu. It supports the necessary *send* and *receive* operations to extract and merge snapshot deltas, respectively. While ZFS is available on platforms other than Solaris, the port's maturity and reliability may discourage administrators from adopting it. We can only add anecdotal evidence to this, by reporting one failed benchmark run due to issues within the ZFS kernel module.

4.1.5 dsync

Our synchronization tool, dsync, differs from `rsync` in two main aspects:

(a) dsync is file-system agnostic because it operates on the block-level. While being file-system agnostic makes dsync more versatile, exactly because it requires no file-system specific knowledge, it also constrains the operation of dsync at the same time. All the file-system level meta-data, e.g., modification time stamps, which are available to tools like, e.g., `rsync`, are unavailable to dsync. dsync implicitly assumes that the synchronization target is older than the source.

(b) Instead of computing block-level checksums at the time of synchronization, dsync tracks the per-block modification status at runtime. This obviates the need for checksum calculation between two subsequent synchronizations.

In addition to the kernel extensions, we implemented two userspace programs: One to extract modified blocks based on the tracked information, called `dmextract`. Extracting modified blocks is done at the synchronization source. The equivalent tool, which is run at the synchronization target, is called `dmmerge`. `dmmerge` reads a stream consisting of block numbers interleaved with block data. The stream is merged with the target block device. The actual network transfer is handled either by `ssh`, if encryption is required, or `nc`, if encryption is unnecessary.

4.2 Setup

Our benchmark setup consisted of two machines: one sender and one receiver. The two machines had slightly different processor and memory configurations. While the sender was equipped with a 6-core AMD Phenom II processor and 12 GiB RAM, the receiver only had a 4-core Phenom II processor and 8 GiB RAM. Both units had a 2 TB spinning disk (Samsung HD204UI) as well as a 128 GB SSD (Intel SSDSC2CT12). Both disks were attached via the Serial ATA (SATA) revision 2. The spinning disk had a 300 GB "benchmark" partition at an outer zone for maximum sequential performance. Except for the ZFS experiments, we formatted the benchmark partition with an ext3 file system. All benchmarks started and ended with a cold buffer cache. We flushed the buffer cache before each run and ensured that all cached writes

are flushed to disk before declaring the run finished. The machines had a PCIe Gigabit Ethernet card which was connected to a gigabit switch. We ran a recent version of Ubuntu (12.04) with a 3.2 kernel. Unless otherwise noted, each data point is the mean of three runs. The synchronized data set consisted of a single file of the appropriate size, e.g., 16 GiB, filled with random data. If a tool interfaced with a block device, we created a loopback device on top of the single file.

4.3 Benchmarks

We used two types of benchmarks, synthetic and realistic, to evaluate dsync's performance and compare it with its competitors. While the synthetic benchmarks allow us to investigate worst case behavior, the realistic benchmarks show expected performance for more typical scenarios.

4.3.1 Random modifications

In our synthetic benchmark, we randomly modified varying percentages of data blocks. Each block had the same probability to be modified. Random modification is a worst case scenario because there is little spatial locality. Real world applications, on the other hand, usually exhibit spatial locality with respect to the data they read and write. Random read/write accesses decrease the effectiveness of data prefetching and write coalescing. Because conventional spinning hard disks have a tight limit on the number of input/output operations per second (IOPS), random update patterns are ill suited for them.

4.3.2 RUBiS

A second benchmark measures the time to synchronize virtual machine images. The tracked VM ran the RUBiS [5] server components, while another machine ran the client emulator. RUBiS is a web application modeled after `eBay.com`. The web application, written in PHP, consists of a web server, and a database backend. Users put items up for sale, bid for existing items or just browse the catalog. Modifications to the virtual machine image resulted, for example, from updates to the RUBiS database.

A single run consisted of booting the instance, subjecting it to 15 minutes of simulated client traffic, and shutting the instance down. During the entire run, we recorded all block level updates to the virtual machine image. The modified block numbers were the input to the second stage of the experiment. The second stage used the recorded block modification pattern while measuring the synchronization time. Splitting the experiment into two phases allows us to perform and repeat them independently.

4.3.3 Microsoft Research Traces

Narayanan et al. [11] collected and published block level traces for a variety of servers and services at Microsoft Research [3]. The traces capture the block level operations of, among others, print, login, and file servers. Out of the available traces we randomly picked the print server trace. Because the print server's non-system volume was several hundred gigabytes in size, we split the volume into non-overlapping, 32 GiB-sized ranges. Each operation in the original trace was assigned to exactly one range, depending on the operation's offset. Further analysis showed that the first range, i.e., the first 32 GiB of the original volume, had the highest number of write operations, close to 1.1 million; more than double of the second "busiest" range.

In addition to splitting the original trace along the space axis, we also split it along the time axis. The trace covers over a period of seven days, which we split into 24 hour periods.

To summarize: for our analysis we use the block modification pattern for the first 32 GiB of the print server's data volume. The seven day trace is further divided into seven 24 hour periods. The relative number of modified blocks is between 1% and 2% percent for each period.

4.4 Results

4.4.1 Random modifications

We start the evaluation by looking at how the data set size affects the synchronization time. The size varies between 1 and 32 GiB for which we randomly modified 10% of the blocks. The first set of results is shown in Figure 2. First of all, we observe that the synchronization time increases linearly with the data set size; irrespective of the synchronization method. Second, `rsync` takes longest to synchronize, followed by `blockmd5sync` on HDD and `copy` on SSD. `copy`, ZFS, and dsync are fastest on HDD and show similar performance. On SSD, dsync and `blockmd5sync` are fastest, with ZFS being faster than `copy`, but not as fast as dsync and `blockmd5sync`. With larger data sizes, the performance difference is more markedly: for example, at 32 GiB dsync, `copy`, and ZFS perform almost identically (on HDD), while `rsync` takes almost five times as long (420 vs. 2000 seconds). To our surprise, copying the entire state is sometimes as fast as or even slightly faster than extracting and merging the differences. Again at 32 GiB, for example, `copy` takes

[3] available at `ftp://ftp.research.microsoft.com/pub/austind/MSRC-io-traces/`

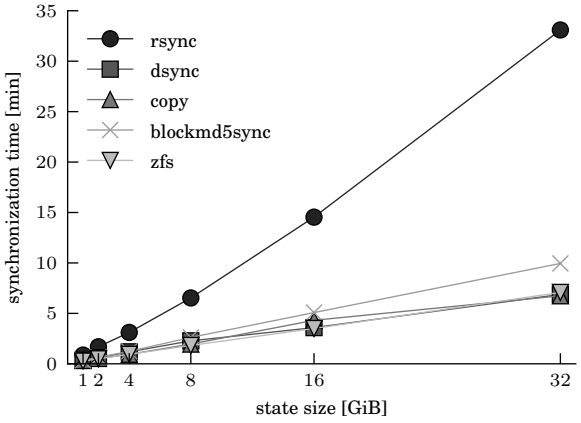
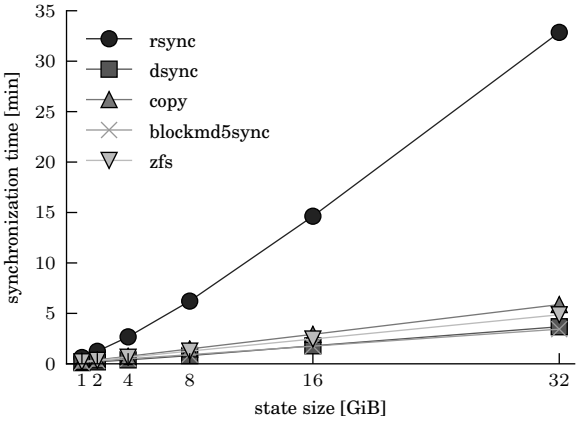
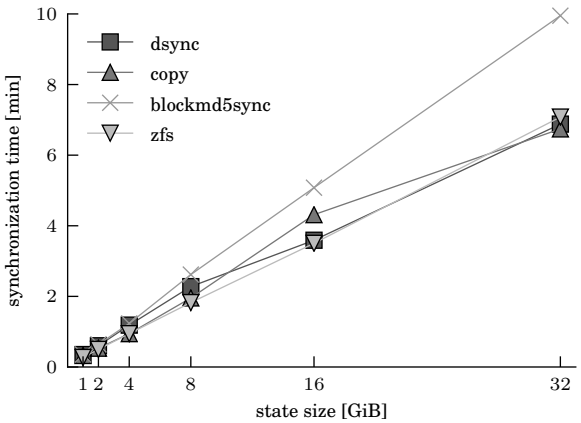


Figure 2: Synchronization time for five different synchronization techniques. Lower is better. Data on the source and target was stored on HDD.





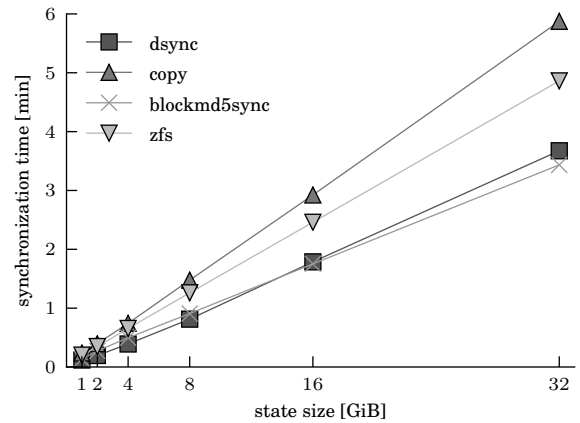


Figure 3: Synchronization time for five different synchronization techniques. Lower is better. Data on the source and target was stored on SSD.

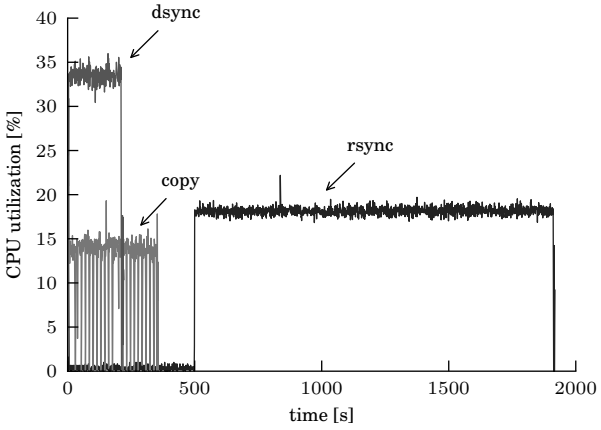


Figure 4: CPU utilization for a sample run of three synchronization tools. 100% means all cores are busy.

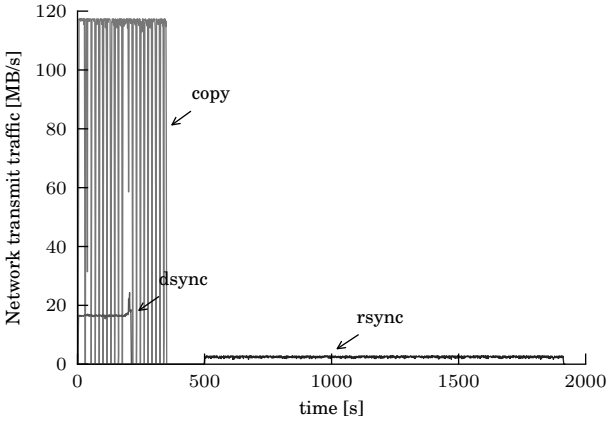


Figure 5: Network transmit traffic on the sender side measured for the entire system. rsync and dsync transmit about the same amount of data in total, although the effective throughput of `rsync` is much lower.

about 400 seconds, compared with 400 seconds for dsync and 420 seconds for ZFS.

We concluded that the random I/O operations were inhibiting dsync performance. Hence, we performed a second set of benchmarks where we used SSDs instead of HDDs. The results are shown in Figure 3. While the increased random I/O performance of SSDs does not matter for `rsync`, its synchronization time is identical to the HDD benchmark, SSDs enable all other methods to finish faster. dsync's time to synchronize 32 GiB drops from 400 s on HDD to only 220 s on SSD.

Intrigued by the trade-off between hard disk and solid state drives, we measured the read and write rate of our drives outside the context of dsync. When extracting or merging modified blocks they are processed in increasing order by their block number. We noticed that the read/write rate increased by up to 10x when processing a *sorted* randomly generated sequence of block numbers compared to the same *unsorted* sequence. For a random but sorted sequence of blocks our HDD achieves a read rate of 12 MB/s and a write rate of 7 MB/s. The SSD reads data twice as fast at 25 MB/s and writes data more than 15x as fast at 118 MB/s. This explains why, if HDDs are involved, `copy` finishes faster than dsync although `copy`'s transfer volume is 9x that of dsync: sequentially going through the data on HDD is much faster than selectively reading and writing only changed blocks.

To better highlight the differences between the methods, we also present CPU and network traffic traces for three of the five methods. Figure 4 shows the CPU utilization while Figure 5 shows the outgoing network traffic at the sender. The trace was collected at the sender while synchronizing 32 GiB from/to SSD. The CPU utilization includes the time spent in kernel and user space, as well as waiting for I/O. We observe that `rsync` is CPU-bound by its single-threaded rolling checksum computation. Up to $t = 500$ the `rsync` sender process is idle, while one core on the receiver-side computes checksums (not visible in the graph). During `rsync`'s second phase, one core, on our 6-core benchmark machine, is busy computing and comparing checksums for the remaining 1400 s (23 min). The network traffic during that time is minimal at less than 5 MB/s. `copy`'s execution profile taxes the CPU much less: utilization oscillates between 0% and 15%. On the other hand, it can be visually determined that `copy` generates much more traffic volume than either `rsync` or dsync. `copy` generates about 90 MB/s of network traffic on average. dsync's execution profile uses double the CPU power of `copy`, but only incurs a fraction of the network traffic. dsync's network throughput is limited by the random read-rate at the sender side.

Even though the SSD's specification promises 22.5k random 4 KiB reads [2], about 90 MiB/s, we are only able to read at a sustained rate of 20 MB/s at the application layer. Adding a loopback device to the configuration, reduces the application layer read throughput by about another 5 MB/s. This explains why dsync's sender transmits at 17 MB/s. In this particular scenario dsync's performance is read-limited. Anything that would help with reading the modified blocks from disk faster, would decrease the synchronization time even further.

Until now we kept the modification ratio fixed at 10%, which seemed like a reasonable change rate. Next we

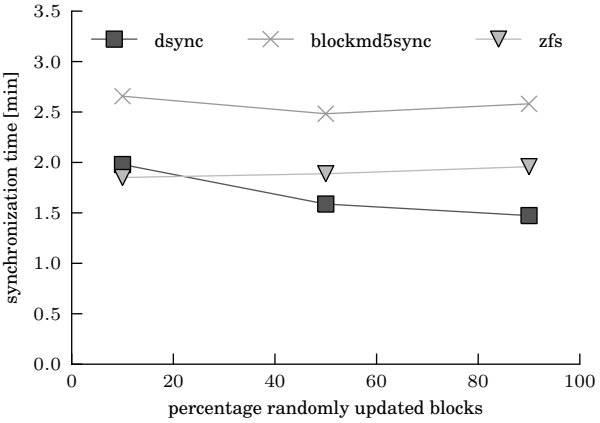


Figure 6: For comparison, `rsync` synchronizes the same data set 6, 21, and 41 minutes, respectively. `copy` took between 1.5 and 2 minutes.

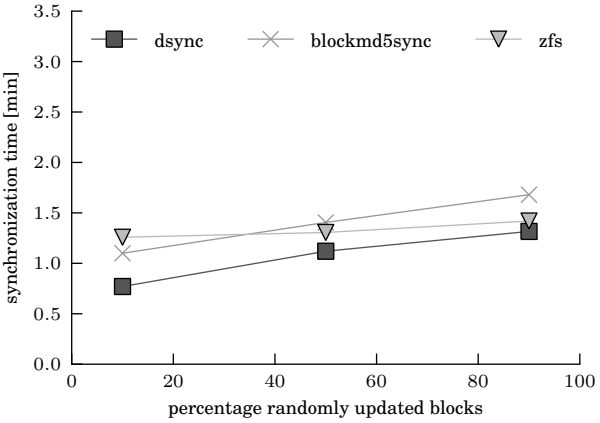


Figure 7: Varying the percentage of modified blocks for an 8 GiB file/device. For comparison, `rsync` synchronizes the same data set in 5, 21, and 41 minutes, respectively. A plain copy consistently took 1.5 minutes.

explored the effect of varying the percentage of modified blocks. The data size was fixed at 8 GiB and we randomly modified 10%, 50%, and 90% percent of the blocks. Figure 6 and 7 shows the timings for spinning and solid-state disks. On HDD, interestingly, even though the amount of data sent across the network increases, the net synchronization time stays almost constant for ZFS and `blockmd5sync`; it even *decreases* for dsync. Conversely, on SSD, synchronization takes longer with a larger number of modified blocks across all shown methods; although only minimally so for ZFS. We believe the increase for dsync and `blockmd5sync` is due to a higher number of block-level re-writes. Updating a block of flash memory is expensive and often done in units larger than 4 KiB [8]. ZFS is not affected by this phenomenon, as ZFS employs a copy-on-write strategy which turns random into sequential writes.

4.4.2 RUBiS results

We argued earlier, that a purely synthetic workload of random block modifications artificially constrains the performance of dsync. Although we already observed a 5x improvement in total synchronization time over `rsync`, the gain over `copy` was less impressive. To highlight the difference in spatial locality between the synthetic and RUBiS benchmark, we plotted the number of consecutive modified blocks for each; this is illustrated in Figure 8.

We observe that 80% of the modifications involve only a single block (36k blocks at $x = 1$ in Figure 8). In comparison, there are no single blocks for the RUBiS benchmark. Every modification involves at least two consecutive blocks (1k blocks at $x = 2$). At the other end of the spectrum, the longest run of consecutively modified blocks is 639 for the RUBiS benchmarks. Randomly updated blocks rarely yield more than 5 consecutively modified blocks. For the RUBiS benchmark, updates of 5 consecutive blocks happen most often: the total number of modified blocks jumps from 2k to 15k moving from 4 to 5 consecutively modified blocks.

Now that we have highlighted the spatial distribution of updates, Figure 9 illustrates the results for our RUBiS workload. We present numbers for the HDD case only because this workload is less constrained by the number of I/O operations per second. The number of modified blocks was never the same between those 20 runs. Instead, the number varies between 659 and 3813 blocks. This can be explained by the randomness inherent in each RUBiS benchmark invocation. The type and frequency of different actions, e.g., buying an item or browsing the catalog, is determined by chance. Actions that modify the database increase the modified block count.

The synchronization time shows little variation between runs of the same method. `copy` transfers the entire 11 GiB of data irrespective of actual modifications. There should, in fact, be no difference based on the number of modifications. `rsync`'s execution time is dominated by checksum calculations. `dsync`, however, transfers only modified blocks and should show variations. The relationship between modified block count and synchronization time is just not discernible in Figure 9. Alternatively, we calculated the correlation coefficient for dsync which is 0.54. This suggests a positive correlation between the number of modified blocks and synchronization time. The correlation is not perfect because factors

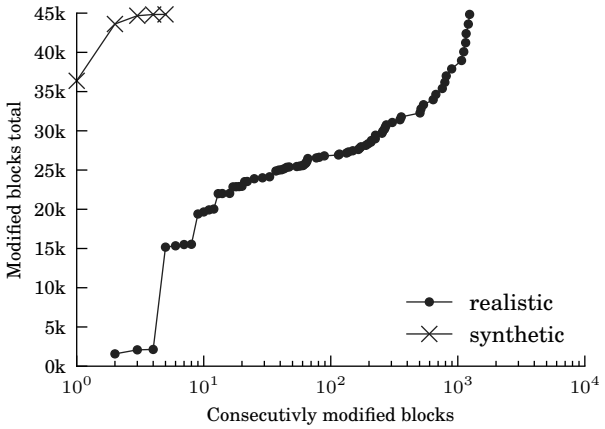


Figure 8: Difference in spatial locality between a synthetic and realistic benchmark run. In both cases 45k blocks are modified. For the synthetic benchmark 80% are isolated individual blocks (36k at x=1). The realistic benchmark shows a higher degree of spatial locality, as observed, for example, by the jump from 2.5k (x=3) to 15k (x=4) blocks.

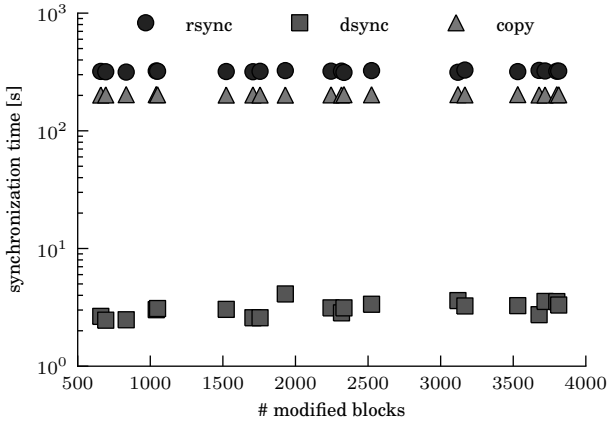


Figure 9: Synchronization time for (a) `copy`, (b) `rsync`, and (c) `dsync`. Block modifications according to the RUBiS workload.

other than the raw modified block count affect the synchronization time, e.g., the spatial locality of updates.

The performance in absolute numbers is as follows: `rsync`, which is slowest, takes around 320 seconds to synchronize the virtual machine disk. The runner up, `copy`, takes 200 seconds. The clear winner, with an average synchronization time of about 3 seconds, is dsync. That is a 66x improvement over `copy` and more than 100x faster than `rsync`. dsync reduces the network traffic to a minimum, like `rsync`, while being 100x faster. Table 2 summarizes these results.

4.4.3 Microsoft Research Traces

In addition to our benchmarks with synchronizing a single virtual machine disk, we used traces from a Microsoft Research (MSR) printer server. The speed with which the different methods synchronize the data is identical across all days of the MSR trace. Because of the homogeneous run times, we only plot three days out of the total seven.

`rsync` is slowest, taking more than 900 seconds (15 minutes) to synchronize 32 GiB of binary data. The small number of updated blocks (between 1-2%) decreases the runtime of `rsync` noticeably. Previously, with 10% updated blocks `rsync` took 35 minutes (cf. Figure 2) for the same data size. `copy` and `blockmd5sync` finish more than twice as fast as `rsync`, but are still considerably slower than either ZFS or dsync. The relative order of synchronization times does not change when we swap HDDs for SSDs (Figure 11). Absolute synchronization times improve for each synchronization method. `blockmd5sync` sees the largest decrease as its performance is I/O bound on our setup: the SSD offers faster sequential read speeds than the HDD, 230 MB/s vs 130 MB/s.

4.5 Discussion

One interesting question to ask is if there exist cases where dsync performs worse than `rsync`. For the scenarios investigated in this study dsync always outperformed `rsync` by a significant margin. In fact, we believe, that dsync will always be faster than `rsync`. Our reasoning is that the operations performed by `rsync` are a superset of dsync's operations. `rsync` must read, transmit, and merge all the updated blocks; as does dsync. However, to determine the updated blocks `rsync` must read *every* block and compute its checksum at the source *and* destination. As illustrated and mentioned in the capture for Figure 4, the computational overhead varies with the number of modified blocks. For identical input sizes, the execution time of `rsync` grows with the number of updated blocks.

The speed at which dsync synchronizes depends to a large degree on the spatial distribution of the modified blocks. This is most visible in Figures 6. Even though the data volume increases by 5x, going from 10% randomly modified blocks to 50%, the synchronization takes *less* time. For the scenarios evaluated in this paper, a simple copy typically (cf. Figure 6, 2) took at least as long as dsync. While dsync may not be faster than a plain copy in all scenarios, it definitely reduces the volume of transmitted data.

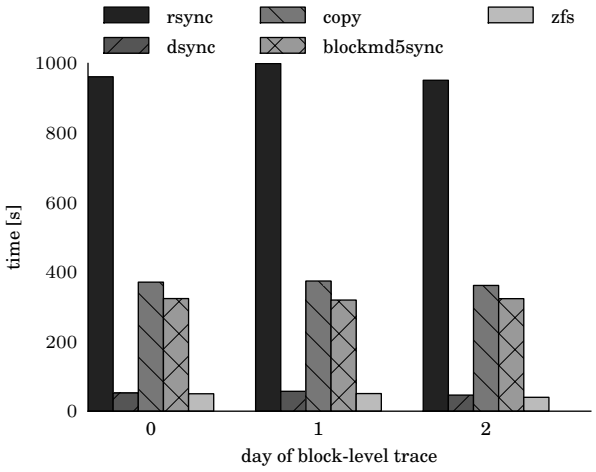


Figure 10: Synchronization times for realistic block-level update patterns on HDDs. Lower is better. The results for the remaining days 3-6 are identical to the first.

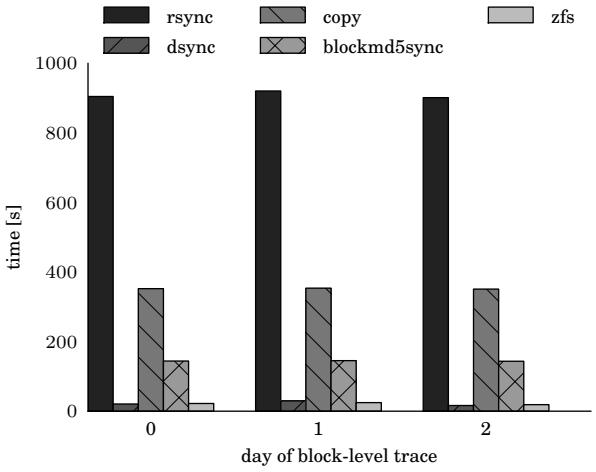


Figure 11: Synchronization times for realistic block-level update patterns on SSDs. Lower is better. The results for the remaining days 3-6 are identical to the first.

Regarding the runtime overhead of maintaining the bitmap used to track changed blocks, we do not expect this to noticeably affect performance in typical use cases. Setting a bit in memory is orders of magnitude faster than actually writing a block to disk.

5 Related work

`lvmsync` [16] is a tool with identical intentions as dsync, but less generic than dsync. While dsync operates on arbitrary block devices, `lvmsync` only works for partitions managed by the logical volume manager (LVM). `lvmsync` extracts the modifications that happened to an LVM partition since the last snapshot. To provide snapshots, LVM has to keep track of the modified blocks, which is stored as meta-data. `lvmsync` uses this meta-data to identify and extract the changed blocks.

File systems, such as ZFS, and only recently btrfs, also support snapshots and differential backups. In ZFS, differential backups are performed using the ZFS *send* and *receive* operations. The delta between two snapshots can be extracted and merged again with another snapshot copy, e.g., at a remote backup machine. Only users of ZFS, however, can enjoy those features. For btrfs, there exists a patch to extract differences between two snapshot states [6]. This feature is, however, still considered experimental. Besides the file system, support for block tracking can be implemented higher up still in the software stack. VMware ESX, since version 4, is one example which supports block tracking at the application layer. In VMware ESX server the feature is called *changed block tracking*. Implementing support for efficient, differential backups at the block-device level, like dsync does, is more general, because it works regardless of the file system and application running on top.

If updates must be replicated more timely to reduce the inconsistency window, the distributed replicated block device (DRBD) synchronously replicates data at the block level. All writes to the primary block device are mirrored to a second, standby copy. If the primary block device becomes unavailable, the standby copy takes over. In single primary mode, only the primary receives updates which is required by file systems lacking concurrent access semantics. Non-concurrent file systems assume exclusive ownership of the underlying device and single primary mode is the only viable DRBD configuration in this case. However, DRBD also supports dual-primary configurations, where both copies receive updates. A dual-primary setup requires a concurrency-aware file system, such as GFS or OCFS, to maintain consistency. DRBD is part of Linux since kernel version 2.6.33.

There also exists work to improve the efficiency of synchronization tools. For example, Rasch and Burns [13] proposed for `rsync` to perform in-place updates. While their intention was to improve `rsync` performance on resource-constraint mobile devices, their approach also helps with large data sets on regular hardware. Instead of creating an out-of-place copy and atomically swapping this into place at the end of the transfer, the patch performs in-place updates. Since their original patch, in-place updates have been integrated into regular `rsync`.

A more recent proposal tackles the problem of *page cache pollution* [3]. During the backup process many files and related meta-data are read. To improve system

Tool	Sync time [s]	State transferred [MB]
`rsync`	950	310
`copy`	385	32768
`blockmd5sync`	310	310
ZFS	42	310
dsync	38	310

Table 2: Performance summary for realistic benchmark.

performance, Linux uses a page cache, which keeps recently accessed files in main memory. By reading large amounts of data, which will likely *not* be accessed again in the near future, the pages cached on behalf of other processes must be evicted. The above mentioned patch reduces cache pollution to some extent. The operating system is advised, via the `fadvise` system call, that pages, accessed as part of the `rsync` invocation, can be evicted immediately from the cache. Flagging pages explicitly for eviction, helps to keep the working sets of other processes in memory.

Effective buffer cache management was previously discussed, for example, by Burnett et al. [4] and Plonka et al. [12]. Burnett et al. [4] reverse engineered the cache replacement algorithm used in the operating system. They used knowledge of the replacement algorithm at the application level, here a web server, to change the order in which concurrent requests are processed. As a result, the average response time decreases and throughput increases. Plonka et al. [12] adapted their network monitoring application to give the operating system hints about which blocks can be evicted from the buffer cache. Because the application has ultimate knowledge about access and usage patterns, the performance with application-level hints to the OS is superior. Both works agree with us on the sometimes adverse effects of the default caching strategy. Though the strategy certainly improves performance in the average case, subjecting the system to extreme workloads, will reveal that sometimes the default is ill suited.

6 Conclusion

We tackled the task of periodically synchronizing large amounts of binary data. The problem is not so much how to do it, but how to do it *efficiently*. Even today, with widespread home broadband connections, network bandwidth is a precious commodity. Using it to transmit gigabytes of redundant information is wasteful. Especially for data sets in the terabyte range the good old "sneakernet" [10] may still be the fastest transmission mode. In the area of cloud computing, large binary data blobs prominently occur in the form of virtual machine disks and images. Backup of virtual machine disks and images, e.g., for fault tolerance, is a routine task for any data center operator. Doing so efficiently and with minimal impact on productive workloads is in the operator's best interest.

We showed how existing tools, exemplified by `rsync`, are ill-suited to synchronize gigabyte-sized binary blobs. The single-threaded checksum computation employed by `rsync` leads to synchronization times of 32 minutes even for moderate data sizes of 32 GB. Instead of calculating checksums when synchronization is requested, we track modifications on line. To do so, we extended the existing device mapper module in the Linux kernel. For each tracked device, the modified block numbers can be read from user-space. Two supplemental tools, called `dmextract` and `dmmerge`, implement the extraction and merging of modified blocks in user-space. We call our system dsync.

A mix of synthetic and realistic benchmarks demonstrates the effectiveness of dsync. In a worst case workload, with exclusively random modifications, dsync synchronizes 32 GB in less than one quarter of the time that `rsync` takes, i.e., 7 minutes vs 32 minutes. A more realistic workload, which involves the synchronization of virtual machines disks, reduces the synchronization time to less than $1/100$th that of `rsync`.

Acknowledgments

The authors would like to thank Carsten Weinhold, and Adam Lackorzynski for their helpful insights into the Linux kernel, as well as Björn Döbel, Stephan Diestelhorst, and Thordis Kombrink for valuable comments on drafts of this paper. In addition, we thank the anonymous LISA reviewers and in particular our shepherd, Mike Ciavarella. This research was funded as part of the ParaDIME project supported by the European Commission under the Seventh Framework Program (FP7) with grant agreement number 318693.

References

[1] URL `http://zfsonlinux.org`.

[2] Intel solid-state drive 330 series: Specification. URL `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-330-specification.pdf`.

[3] Improving Linux Performance by Preserving Buffer Cache State. URL `http://insights.oetiker.ch/linux/fadvise/`.

[4] N. Burnett, J. Bent, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Management. In *Proceedings of USENIX Annual Technical Conference*, pages 29–44, 2002.

[5] OW2 Consortium. RUBiS: Rice University Bidding System. URL http://rubis.ow2.org.

[6] Jonathan Corbet. Btrfs send/receive. URL http://lwn.net/Articles/506244/.

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 3rd edition edition, 2009.

[8] Michael Cornwell. Anatomy of a Solid-state Drive. *Queue*, 10 (10), 2012.

[9] Pawel Jakub Dawidek. Porting the ZFS file system to the FreeBSD operating system. *AsiaBSDCon*, pages 97–103, 2007.

[10] Jim Gray, Wyman Chong, Tom Barclay, Alexander S. Szalay, and Jan vandenBerg. TeraScale SneakerNet: Using Inexpensive Disks for Backup, Archiving, and Data Exchange. *CoRR*, cs.NI/0208011, 2002.

[11] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Transactions on Storage*, 4(3), 2008.

[12] D. Plonka, A. Gupta, and D. Carder. Application Buffer-Cache Management for Performance: Running the World's Largest MRTG. In *Large Installation System Administration Conference*, pages 1–16, 2007.

[13] D. Rasch and R. Burns. In-Place Rsync: File Synchronization for Mobile and Wireless Devices. In *Proc. of the USENIX Annual Technical Conference*, 2003.

[14] David Rosenthal. Keeping Bits Safe: How Hard Can It Be? *Queue*, 8(10), 2010. URL http://queue.acm.org/detail.cfm?id=1866298.

[15] P.J. Salzman, M. Burian, and O. Pomerantz. The Linux Kernel Module Programming Guide. 2001. URL http://www.tldp.org/LDP/lkmpg/2.6/html/.

[16] Mick Stanic. Optimised synchronisation of lvm snapshots over a network. URL http://theshed.hezmatt.org/lvmsync/.

[17] A. Tridgell and P. Mackerras. The rsync Algorithm. Technical Report TR-CS-96-05, Australian National University, 1996.