PersonalRAID: Mobile Storage for Distributed and Disconnected Computers

Sumeet Sobti* Nitin Garg* Chi Zhang* Xiang Yu*
Arvind Krishnamurthy† Randolph Y. Wang*

Abstract

This paper presents the design and implementation of a mobile storage system called a PersonalRAID. PersonalRAID manages a number of disconnected storage devices. At the heart of a PersonalRAID system is a mobile storage device that transparently propagates data to ensure eventual consistency. Using this mobile device, a PersonalRAID provides the abstraction of a single coherent storage name space that is available everywhere, and it ensures reliability by maintaining data redundancy on a number of storage devices. One central aspect of the PersonalRAID design is that the entire storage system consists solely of a collection of storage logs; the log-structured design not only provides an efficient means for update propagation, but also allows efficient direct I/O accesses to the logs without incurring unnecessary log replay delays. The PersonalRAID prototype demonstrates that the system provides the desired transparency and reliability functionalities without imposing any serious performance penalty on a mobile storage user.

1 Introduction

As disk density continues to grow at a phenomenal annual rate of 100% [3], the cost, form factor, and capacity of stable storage continues to improve dramatically. One consequence of these dramatic technological advances is the emergence of highly compact secondary storage, which can be seamlessly integrated into devices of all shapes and forms. As technology continues to improve, as decentralization is carried to its logical next step, and as traditionally analog information is increasingly being turned into digital representations, it is not unreasonable to conjecture that such mobile storage may become a dominant form of storage in the near future, especially for personal user data, subsuming conventional disks enshrined in machine rooms.

Figure 1: An IBM Microdrive. On the left is a 1 GB Microdrive shown in its packaging. On the right is the same drive shown open with a U.S. quarter. (Courtesy of IBM. Unauthorized use not permitted.)

Unfortunately, as mobile storage flourishes, high-performance universal network connectivity may still not be available everywhere. At any instant, only a small number of devices may be connected to each other; and a mobile storage user cannot always count on an omnipresent high-quality connectivity to a centralized storage service. A mobile storage solution that does not rely solely on network connectivity for managing a collection of distributed (and possibly disconnected) devices needs to be found. We also share the belief that a user's attention is a precious resource that the system must carefully optimize for, and a central goal of the system is to ease the management of these disconnected devices.

We identify three important desirable features of such a mobile storage solution. (1) *The availability of a single coherent name space.* A user who owns a number of storage devices should not be burdened with the chore of hoarding needed data and propagating updates manually. Ideally, even when these devices are permanently disconnected, the user still sees a single coherent space of data regardless where she is and regardless which device she uses. The user should not have to modify her existing applications to enjoy these benefits. (2) *Reliability.* The view that only centralized servers provide reliability guarantees and mobile devices are inferior second-class citizens, whose data is expendable, is not al-

*Department of Computer Science, Princeton University, {sobti,nitin,chizhang,xyu,rywang}@cs.princeton.edu.
†Department of Computer Science, Yale University, arvind@cs.yale.edu.
This work is supported in part by IBM. Wang is supported by NSF Career Award CCR-9984790 and Krishnamurthy is supported by NSF Career Award CCR-9985304.

ways acceptable: the frequency and duration of periods where a mobile storage device is disconnected from a central server can be significant enough that one must have some degree of confidence over the reliability of data. (3) *Acceptable performance.* The provision of the transparency and reliability features listed above should not impose a significant overhead. Ideally, the user of the mobile storage system should always be able to enjoy a level of performance that is close to that of the local storage.

In this paper, as a first step, we describe a mobile storage system called a *PersonalRAID* that is designed to support a collection of disconnected and distributed personal computers. In addition to these end hosts, central to the PersonalRAID design is a portable storage device, such as the 1 GB IBM Microdrive (Figure 1) used in our prototype. We call this device the *Virtual-A* (VA). The VA is named so because: (1) it is analogous to a conventional removable storage device (which is often named drive A on Windows PCs), and (2) it provides the illusion of a storage device whose capacity is far greater than the device's physical capacity. The VA allows the PersonalRAID system to achieve the three goals enumerated above: (1) The VA transparently propagates updates among disconnected hosts to ensure eventual consistency. It helps provide a single storage name space that is transparently available on all hosts. It supports existing file systems and applications without modification. (2) The VA provides temporary redundancy before data is propagated to multiple end hosts so that the PersonalRAID system can tolerate any single-device loss. (3) The PersonalRAID system intelligently chooses between a local disk and the VA device to satisfy I/O requests to mask propagation delays and minimize overhead.

The central aspect of PersonalRAID design is its use of a distributed log-structured design: the collection of distributed logs *is* the storage system: there is no other permanent structure hosting the data. This design not only allows PersonalRAID to propagate updates among the logs throughout the system efficiently, it also allows a user to satisfy her I/O requests directly from the logs without having to wait for propagations to complete. We have implemented a prototype PersonalRAID system. Our experiments demonstrate that the system achieves the transparency and reliability functional goals without imposing any serious performance penalty.

The rest of the paper is structured as follows. Section 2 describes the user experience and the main operations of a PersonalRAID system. Section 3 presents the rationale and the details of the log-structured design of the system. Section 4 describes our prototype PersonalRAID implementation. Section 5 details the experimental results. Section 6 compares PersonalRAID to a number of related systems. Section 7 concludes.

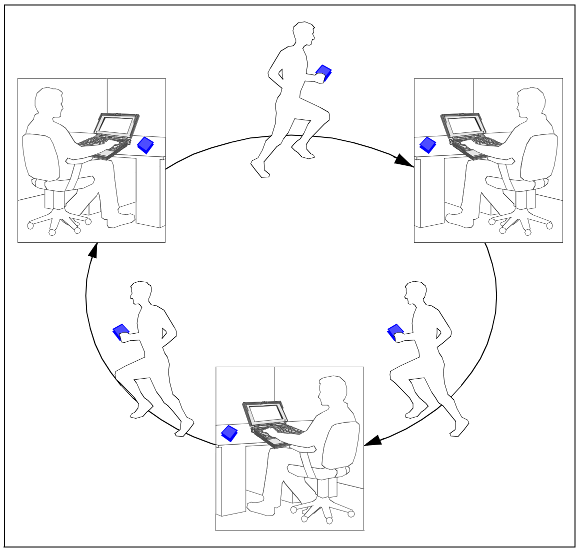

Figure 2: A usage scenario of PersonalRAID. The mobile user carries the Virtual-A device as the user travels among disconnected computers (such as a machine at home, a machine in the office, and a laptop on the go). The PersonalRAID system provides the illusion of a single name space, and it also ensures data reliability.

2 Functionalities

The PersonalRAID system manages a number of disconnected storage devices where the mobile user desires a single name space on all of them. The mobile VA device is instrumental in bringing this about. It is generally not a good idea to rely exclusively on the mobile storage device alone due to its capacity, performance, and reliability limitations; instead, the device needs to be an integral part of a PersonalRAID system.

The VA accompanies the user wherever she goes (Figure 2). With current technology, a few gigabytes can be packaged in the form factor of a credit card (such as the Kingston 5 GB DataPak PC Card Type II Hard Drive) or a wrist watch (such as the IBM 1 GB Microdrive). The VA can communicate with a host computer via various forms of connectivity (such as PCMCIA, USB, or Bluetooth). As long as the VA is present, the user "sees" her up-to-date large home directory regardless where she is and which computer she is using. The user never needs to perform manual hoarding or manual propagation of data; and the loss or theft of any single device does not result in data loss. PersonalRAID mainly targets personal usage scenarios. (We ad-

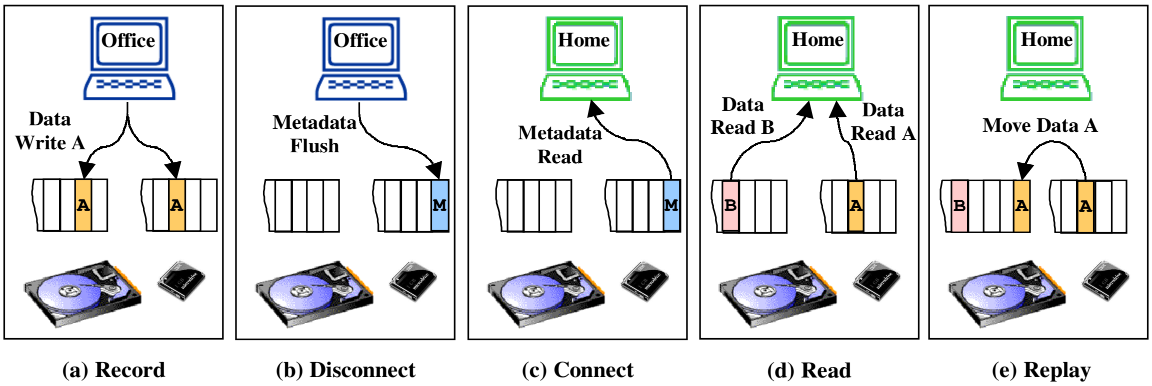

Figure 3: PersonalRAID operations. The roles of the home and office computer disks are reversed as the user creates new data at home, which must be propagated to the office.

dress concurrent updates in Section 3.8.1.)

Figure 3 depicts in greater detail the PersonalRAID operations that synchronize the contents of several host computer disks. (a) In this example, when in the office, the VA passively observes the I/Os performed to the office computer disk and incrementally *records* the newly written data. The office computer disk is called the *source* of this data. (b) When the user is about to leave the office and *disconnects* the VA, the system flushes some metadata so that an inventory of the VA's contents is placed on it. (c) The user then takes the VA home and *connects* the device to her home computer. The system reads the metadata, and file system operations can occur immediately following connection. (d) After connection, the system *reads* from either the VA or the home computer disk to satisfy user requests. We call the I/O events that have occured between a pair of connection and disconnection events a *session*. (e) Possibly in the background, PersonalRAID synchronizes the contents of the disks by *replaying* some of the updates, which were recorded on the VA earlier in the office, to the home computer disk. The home computer disk is called the *destination*. Only after the latest updates are reflected on all the host disks do we remove the copy of the data from VA. (This invariant can be a problem if some hosts in the system are only infrequently visited by the user, and the VA device is not large enough to hold all the unpropagated data.) Note that it is not necessary to replay all the new data to a destination device in a single session—the user may choose to disconnect from the home computer at any time. As the user creates new data at home and the VA records it for later replaying to the office disk, the roles of the two end hosts are reversed: the home disk becomes the source and the office disk becomes the destination. Note that we are maintaining an invariant: a copy of any data resides on at least two devices. This invariant allows the system to recover from any single-device loss.

Our current implementation requires the VA device to be present when the file system is being accessed; this is consistent with the most common single-user case that PersonalRAID targets. We do not currently handle updates on a host that is disconnected from the VA. Such updates can be potentially conflicting. In Section 3.8.1, we discuss ways in which the current system could be extended to address these limitations. Another limitation is that the incorporation of a new host into the system requires a heavy-weight recovery operation, as discussed in Section 3.6.

For simplicity, our current system addresses only disconnected computers. We are researching extensions of the system that can exploit weak or intermittent network connectivity when it is present. We expect the mobile storage device and the weak connectivity to complement each other in such situations.

3 Design

The design of the PersonalRAID should satisfy the following requirements:

- Recording should not impose excessive overhead that may interfere with normal I/O operations.
- During disconnection, the user should not be forced to wait for long before the VA can be safely removed.
- During connection, the user should not be forced to wait for long before she is allowed to perform I/O operations.
- Replaying should not impose excessive overhead that may interfere with normal I/O operations.
- Replaying should proceed quickly so that the disk space on the VA can be quickly freed up for future I/Os.

3.1 Naive Design Alternatives

A simple solution is the following. At the end of the work day in the office, for example, the system copies all the content that was updated during the day to the mobile disk. After the user reaches home, the system copies this content to the home computer disk. While relatively simple to implement, this approach has some serious disadvantages: before leaving the office, the user is forced to wait for the entire set of newly modified data to be copied from the source to the mobile device; and after reaching home, the user must wait for the entire mobile device content to be copied to the destination before she is allowed to access the file system. With mobile storage devices that can store gigabytes of data, and with potentially lengthy intervals spent at one computer before moving onto another, the latency may become intolerable.

One possible improvement is to incrementally copy newly generated data from the source disk to the mobile disk in the background instead of allowing the new data to accumulate. This improves the disconnection time, but it does not address the long connection latency–the user still must wait for the entire propagation to complete before she can proceed to normal I/O activities.

To address these disadvantages, one realizes that the PersonalRAID needs to be a file system or storage system solution which can transparently decide which device to access for a particular piece of data: this is necessary, for example, if the user desires to access the data on the mobile device after connection but before it is propagated to the destination device. With this requirement in mind, let us consider a second alternative: during recording, the system mirrors a portion of the source device Unix File System (UFS) on the mobile device. After connection, while background replaying occurs, the user may transparently access the mirrored portion of the UFS on the mobile device.

While this second design alternative may improve the disconnection and connection latency, the choice of mirroring a portion of the UFS on the mobile device may not be a wise decision for best recording and replaying performance. First, during recording, UFS updates may incur a large number of small synchronous disk writes. The problem is made worse when the mobile storage devices typically do not possess the best latency characteristics so it is difficult to mask the extra latency by overlapping the I/Os to different devices. This situation is especially unfortunate when one realizes that the synchronous mirroring on the mobile device is unnecessary–data is already made persistent on the source device.

Second, replaying from a partial UFS mirror on the mobile device to the destination device UFS is also inefficient, because unnecessary disk head movement occurs. Slow replaying, in turn, has several negative repercussions: (1) slow replaying interferes with normal user I/O activity; (2) slow replaying may cause the mobile device to fill up; and (3) slow replaying prevents the user from taking advantage of the potentially faster destination device by forcing the user to continue to use the slower mobile device for reads.

3.2 Log-Structured Organization

The analysis of the naive design alternatives shows that a good PersonalRAID design should have at least these two properties: (1) the mobile device should be an integral part of a storage or file system so that the user can transparently read/write the device without incurring long connection/disconnection latency; (2) the transfer of data onto/off the mobile device should take place in a fashion that avoids incurring the intrinsic latency bottleneck of disks.

One possible design that naturally satisfies these requirements is to have some variant of a log-structured file system (LFS) [12] on both the VA and the host disks. During recording, data is buffered in large memory *segments*; these memory buffers prevent overwritten data from ever reaching the disks and large segment-sized writes are efficient. Disconnection is analogous to a graceful LFS shutdown and connection is analogous to an LFS recovery, both of which mainly involve metadata operations that are relatively efficient. Note that disconnection must also flush any dirty data segments. Fast replaying is possible because the system transfers data at large segment-sized granularity that fully utilizes the VA and host disk bandwidth. Furthermore, during replaying, as the system reads live data from the VA and writes them to the destination device, large extents of empty segments are generated on both the VA and the destination device; therefore, replaying and segment cleaning in effect become an integral one.

3.3 PersonalRAID Data Structures

While it is possible to build a PersonalRAID system at the file system level by modifying an LFS to adapt to multiple storage devices, we have elected to construct the PersonalRAID by extending the design of a Log-Structured Logical Disk (LLD) [1]. A *logical disk* behaves just like a normal disk from the point of view of a file system: it allows the file system to read and write *logical disk addresses*. A particular

	Local Disk	Virtual-A	Location
Segment Summary	PA → LA, t	PA → LA, t	Disk
Checkpoint	LA → PA	LA → PA, bitmap	Disk
Map	LA → PA	LA → PA, state	Memory

PA	physical device address.
LA	logical (virtual) device address.
t	time stamp (global counter).
bitmap	one bit (b_i) per host, b_i=1 if host i needs propagation.
state	4 bits of state information for a block.
s_0	1 iff the block needed to be propagated to the current host at the beginning of the current session.
s_1	1 iff the block needed to be propagated to any of the other hosts at the beginning of the current session.
s_2	1 iff the block has been propagated from the VA to the current host in the current session.
s_3	1 iff the block has been overwritten in the current session.

Figure 4: Details of the main PersonalRAID data structures.

implementer of a logical disk, however, can choose to map these logical addresses to physical addresses in a way that she sees fit. A log-structured logical disk maps logical addresses that are written together to consecutive physical addresses, effectively accomplishing the goals of an LFS. An LLD can support existing file systems with little or no modification and it is typically easier to implement an LLD than an LFS. The PersonalRAID system makes a single consistent logical disk available on all the participating hosts, despite the lack of *any* network connection between any of them.

The data structures on each device of a PersonalRAID are not unlike those of a conventional LLD. In a conventional LLD, the disk is structured as a segmented log. Data blocks are appended to the log as the LLD receives write requests from the file system. Each write request is assigned a *unique* time stamp from a monotonically-increasing global counter. Each segment contains a segment summary, which has the logical address and the time stamp of each data block in the segment. The segment summary aids crash recovery of the in-memory logical-to-physical (L-to-P) address mapping. The L-to-P mapping is checkpointed to the disk during graceful shutdown.

Figure 4 shows in greater detail the three main PersonalRAID data structures: the *segment summary*, the *checkpoint*, and the *in-memory map*. For each of the three data structures, there is one version for the host local disk and there is another version for the VA. The version for the local disk is essentially the same as that of a conventional LLD. The VA version is augmented with some additional information.

Each entry of the L-to-P mapping in the VA checkpoint is augmented with a bitmap: one bit (b_i) per host, and $b_i = 1$ if the block for this logical address needs to be propagated to host i. In a PersonalRAID system, the contents of the VA device are *defined* to be the set of data blocks that still need to be propagated to some host. Thus, a block is evicted from the VA device when it has reached each host in the system. In other words, the L-to-P entry for a particular logical address is marked null when the corresponding bitmap contains 0 in all positions. The VA checkpoint is also home to a global counter. All write operations in the system are assigned a unique time stamp from this counter.

Each entry of the VA in-memory map is augmented with a state field that consists of four bits (s_{0-3}): this field is a summary of what *needed* to happen to this block at the beginning of the current session in terms of propagation (s_{0-1}), and what *has* happened to this block on this host in the current session in terms of replaying and recording (s_{2-3}).

When stored as a table, each L-to-P map consumes 4 bytes per logical address. Thus, assuming a block size of 4 KB, the L-to-P map needs 1 MB for each GB of logical address space. The total size of the bitmap fields in the VA checkpoint is MN bits, where N is the number of logical addresses and M is the number of hosts in the system. Thus, the bitmaps need 32KB of VA disk space per host for each GB of logical address space. The state fields in the in-memory data structure for the VA need 128KB for each GB of logical address space. Observe, however, that at any time, most L-to-P entries and the state and the bitmap fields for the VA will be null, since the size of the VA will typically be much smaller than the size of the logical address space. Thus, more compact representations of the VA data structures are possible.

We close this section by making the following observation about the PersonalRAID data structure design. Each device in a PersonalRAID is self-contained in that the physical addresses within the data structures for this device all point to locations within this device. If a logical data block is not found on this device, the pointer to it in the L-to-P mapping of this device is null; in this case, a corresponding pointer on a device that does contain the block points to a true location of the block. The union of all the valid pointers on all devices constitutes the whole logical-to-physical map.

3.4 PersonalRAID Operations

In this section, we describe the various PersonalRAID operations in detail. Figure 3 also illustrates how these operations interact with the underlying log on each device.

3.4.1 Recording

During recording (a), the system appends a newly created logical block to two logs: one on the source device and the other on the VA. The logical address of the newly created data is recorded in each of the two segment summaries along with the latest timestamp. The in-memory map of each device is also updated to reflect the latest locations of the data block. In addition, we set $s_3 = 1$ in the state field of the VA in-memory map to mark the creation event during this session.

3.4.2 Disconnection and Crash Recovery

During disconnection (b), first the file system on the PersonalRAID logical disk is unmounted. Unmounting flushes all the dirty file-system buffers to the PersonalRAID logical disk, and it also indicates to the user that the file system on the local host is unusable when the VA is not connected to it. Then, a graceful shutdown is performed on both the local disk and the VA. For the local disk, we simply write the in-memory map to its checkpoint region. (If the local host is not powering down after disconnection, then the system can choose to keep the map in memory—this optimization can reduce the connection time for the next session on this host. See Section 3.4.3.)

The VA checkpoint region contains the VA L-to-P map and the bitmap fields. Thus, for the VA, in addition to flushing the map, we must read the bitmap fields of the old checkpoint into memory, compute the new bitmap fields using the old bitmap fields and the state fields of the in-memory map, and write a new checkpoint back to the VA. If $s_3 = 1$ (*written* block), we set the bitmap to reflect the need of propagating this block to all other hosts in the system. Otherwise, if $s_2 = 1$ (*propagated* block), we clear the corresponding bit for this host in the bitmap but retain the values of the remaining bits. We also store the latest timestamp in the VA checkpoint; this timestamp marks the end of the current session and the beginning of the next session. To avoid corrupting the old checkpoint in case a crash occurs in the middle of a checkpoint operation, we maintain two checkpoint regions for each device and alternate between them.

A crash is a special case of disconnection. In a conventional LLD, the goal of crash recovery is to reconcile the contents of the segment summaries and the checkpoint to make them consistent with each other. In the PersonalRAID system, however, the crash recovery process also needs to make the local disk and the VA *mutually consistent* and to restore the PersonalRAID invariants. Note that the PersonalRAID system requires that the recovery process be completed at the crash site before the user moves to another host.

The first PersonalRAID invariant to restore is that all data blocks written in the past (unfinished) session must be present on both the local disk and the VA disk. Because the flushes to the two disks are not synchronized, one of them might have fallen behind the other in terms of receiving the most recent writes. Thus, the crash recovery process might need to propagate data blocks from one disk to the other.

The other PersonalRAID invariant to restore is that the bitmaps in the VA checkpoint must correctly reflect the state of the system in terms of propagation of writes. Blocks written or propagated during the past session might have made the old bitmaps inconsistent. Such blocks can be identified by comparing the time stamps in the segment summaries with the time stamp in the old VA checkpoint. Having identified the written and propagated blocks, bitmaps are updated in the VA checkpoint as in the case of a normal disconnection described above. If all bits in a bitmap are clear, we must have performed all the necessary propagations and we can safely discard this block from the VA by nullifying the logical-to-physical mapping for this block on the VA. We also store the latest timestamp found during the segment summary scan in the VA checkpoint to mark the end of this session.

3.4.3 Connection and Reading

During connection (c), the system needs to initialize the in-memory maps by reading the checkpoints. If the host that the VA is connecting to is powering up, the system needs to initialize the local disk in-memory map by reading the local disk checkpoint, just as LLD does. In a similar fashion, the system reads the VA checkpoint to initialize the VA in-memory map. The difference between the two maps is that the system also needs to calculate the state field of the in-memory map for the VA: s_0 and s_1 are set based on the bitmap stored in the checkpoint (s_0 is set equal to the current host's bit and s_1 is set to the inclusive OR of all other bits), while s_2 and s_3 are cleared. And if $s_0 = 1$, the system concludes that the local disk contains an obsolete copy, and nullifies the logical-to-physical mapping in the local disk in-memory map. Finally, the system also reads the disconnection timestamp in the VA checkpoint to initialize the current timestamp.

As soon as connection completes, the system is ready to accept I/O requests. To service a read request (d), the PersonalRAID looks up the in-memory maps for valid logical-to-physical mappings to decide which device holds the most recent copy of a logical block. In the event that a fresh copy resides on multiple devices, the system is likely to favor the local disk, which is typically faster than the VA, although load-balancing opportunities exist. To service a write request, the PersonalRAID records the new block by appending it to both logs as described earlier.

3.4.4 Replaying

Possibly in the background, the system performs replaying (e). A great deal of synergy exists between PersonalRAID and log-structured storage as replaying is integrated with segment cleaning on the VA device. The system checks the s_0 bit in the VA in-memory map to identify the live data on the VA that is yet to be propagated to the destination device. It then reads live data from the VA and appends it to the log on the destination device. The timestamp of the newly propagated block inherits that of the VA block, which the system reads from the corresponding VA segment summary. The s_2 bit is set in the state field. Next, the system checks s_1 to determine whether this block needs to be propagated to other hosts in the system. If $s_1 = 0$, the data block must have been propagated to all hosts in the system and the block can be safely removed from the VA. If we attempt to populate a segment with blocks of identical propagation bitmaps, then we are likely to harvest free segments as their blocks are freed simultaneously. On the other hand, if $s_1 = 1$, we must retain the block on the VA for further propagation to other hosts in the system and we have two options: we can either leave the blocks in place on the VA in the hope that they may be deleted in the future to render their cleaning unnecessary, or append them to the end of the VA log since we have already incurred the cost of reading them into memory. In the latter case, we have again accomplished segment cleaning on the VA as a byproduct of replaying.

In a conventional LLD, the segment cleaning algorithms mainly aim to maximize the number of free segments generated per unit of cleaning I/O. This is typically achieved by cleaning segments that are relatively cold and have low utilization [12]. (Recall that a piece of data is said to be *cold* if it is unlikely to be overwritten in the near future, and *utilization* of a segment is a measure of the amount of live data in it.) In a PersonalRAID system, however, segment cleaning on the VA device becomes more complicated due to its integration with replaying. Instead of (or in addition to) coldness and utilization of segments, a good cleaning strategy might need to take into account the state of propagation of blocks. For example, at any time, a live block on the VA might need to be propagated (1) to the local host, but not to any remote host, or (2) to some remote host, but not to the local host, or (3) to the local host as well as some remote host. A cleaning strategy might prefer to clean one type of blocks before others depending on how aggressively it wants to replay blocks or generate free segments.

In this section, we examine the log operations during the recording, disconnection, connection, reading, and replaying phases of the PersonalRAID. We note two features of the PersonalRAID design. One feature is that the map information on each device is self-contained: all the valid pointers in a device point to locations within the same device. This feature allows data movement within one device or across a subset of the partially connected devices to be performed independently of other devices. The second feature of the PersonalRAID design is the potential for exploiting I/O parallelism: the recording, reading, and replaying phases can overlap I/Os to the VA and local disks to mask some of the I/O latency and balance load.

3.5 Recovering from Device Losses

We first discuss how to recover from the loss of a host computer disk. Then we discuss how to recover from the loss of the VA device itself.

3.5.1 Recovering from Host Disk Loss

This scenario is the simpler case to handle. We take the VA to a surviving host. The union of the contents of the local disk on this host and the VA gives the entire content of the PersonalRAID. First, we completely synchronize the local disk and the VA by replaying all those blocks whose latest versions are present on the VA but not on the local disk. Thus, at the end of this phase, the bit for this host in all the bitmaps is 0. Then, we create a physical mirror of the local disk onto a new disk (using a Unix utility like `dd`, for example). The new disk is brought to the accident site to replace the lost disk. The bitmaps on the VA are updated to reflect the fact that the restored disk now has all the PersonalRAID data.

3.5.2 Recovering from VA Device Loss

The more complex case is when the VA device itself is lost. There are two pieces to be reconstructed. The first piece is the metadata, which consists of the bitmaps and the current value of the global counter. The other is the set of actual data blocks that were lost with the VA device. Recall that the contents of the VA are *defined* to be the set of data blocks that must be propagated to some host in the system. Thus, to reconstruct the data part, we may need to visit all the hosts in the system.

A simple recovery method is this. We visit all the hosts in the system twice. The goal of the first tour is to construct the metadata part by scanning the segment summaries on each host and comparing time stamps. In the second tour, we simply visit each host to copy the required data blocks onto a new VA device. Note that this tour does not need to scan the segment summaries on the local disks—the L-to-P mapping is sufficient to locate the required data.

An unpleasant aspect of this reconstruction approach is that it requires one to visit each host twice. To eliminate the first tour, we can make a copy of the bitmaps and the global counter on the local disk during the disconnection process. When the VA is lost, the user must retrieve the bitmaps and the value of the global counter from the host where she most recently disconnected. The bitmaps allow the system to identify the hosts on which the latest copy of a VA logical block is stored. The user now needs to complete only one tour to reconstruct the VA content. The price one pays for this simpler approach is the extra time and space spent during disconnection to write the bitmaps to the local disk, although it is likely that one should be able to overlap the bitmap flush time on the local disk with the slower VA checkpoint time, and the space consumed by the bitmaps is insignificant.

This VA reconstruction approach even works if one encounters a combination of a crash and the loss of the VA. In this unfortunate scenario, the user retrieves the old bitmaps from the computer that she visited last and then comes back to the crashed computer to complete the crash recovery process as described in Section 3.4.2. At the end of this crash recovery process, the system has recovered the lost bitmaps and she can start the reconstruction tour to reconstruct the lost VA data. To avoid having to go back to the computer that she visited last in this scenario, we can make a copy of the VA checkpoint bitmaps on the local disk during the connection process (as well as the disconnection process).

The VA reconstruction processes described so far require the user to complete at least one tour of all the hosts. To further reduce the number of hosts that one must visit after a VA loss, one can periodically replay some blocks to *all* the hosts to eliminate these blocks from the VA or periodically replay *all* VA content to some host. The latter technique is effectively same as making a copy of the VA content on that host.

3.6 Reconfiguration

Reconfiguration of a PersonalRAID in the form of removing or adding hosts is relatively simple. To remove a host from the system, all the system has to do is to reformat the VA checkpoint to remove a bit from each bitmap. Recall that the bitmaps record which hosts in the system need to receive propagations. The checkpoint reformat allows the system to discard data from the VA, no longer propagating it to the removed host. To further simplify this process, the system can simply record at the beginning of the checkpoint which bits in the bitmaps are still considered *active*; so the non-active bits are not considered in the algorithm.

Adding a host to the PersonalRAID is essentially the same as recovering from the loss of a host disk. The only difference is the reformatting of the VA checkpoint to add a bit to each bitmap. Again, to further simplify the process, the system can simply *activate* a previously allocated bit. This bit is set to 0 in each bitmap since at the end of recovery (see Section 3.5.1), the recovered host has all the PersonalRAID data.

3.7 Virtual VAs

So far in our discussion, we seem to have assumed that the Virtual-A has to be a physical mobile storage device such as the IBM Microdrive. This assumption is not necessary: the Virtual-A can in fact be backed by a file, a local disk partition, or even a network connection. We call such a virtual backing device a *Virtual Virtual-A* (or a VVA or a $\mathrm{V}^2\mathrm{A}$).

One possible use for a local disk-based VVA is to use it instead of a mobile storage device to perform recording. Because mobile storage devices do not necessarily have the best performance characteristics, recording to a VVA can be more efficient. Of course, to transport the data, we still must copy the contents of a VVA to a mobile device-based VA (and sometimes, vice versa). One possible solution for avoiding the long copying latency is to allow asynchronous copying to the mobile device to occur in the background. Although the log-structured design of the VA organization allows the PersonalRAID to buffer a large amount of data in memory and delete overwritten data before it reaches the device, the amount of buffering is still limited by the available memory. A VVA that asynchronously copies to a VA essentially allows unlimited buffering. Another use of a VVA is to make a copy of the VA for the purpose of reconstructing a lost VA (as described in Section 3.5.2).

To efficiently implement a local disk-backed VVA, one in fact does not need to physically and separately store the data blocks that can already be found on the local disk of this host: the mapping information is all that is needed. Because we do not have to physically store separate copies for a VVA's data in this case, we call this a *Virtual VVA* (or a $\mathrm{V}^3\mathrm{A}$).

3.8 Limitations and Extensions

In this section, we describe a number of limitations of the PersonalRAID system described so far. Some of these are the topics of our continued research and we discuss possible approaches of addressing them.

3.8.1 Concurrent Updates

To simplify the discussion so far, we have made a conscious design decision of not addressing concurrent updates. We believe that this is an acceptable choice for the use cases that we are targeting: PersonalRAID, being "personal", is designed for a single user to control a number of distributed and disconnected personal storage devices; these storage devices do not receive updates concurrently simply because we do not allow the user to be at multiple sites simultaneously. There is, however, nothing intrinsic in the current PersonalRAID design that prohibits us from addressing concurrent updates.

Indeed, there are legitimate personal use cases where concurrent updates arise naturally. For example, a networked office computer can continue to receive email after the user has disconnected the VA and has gone on vacation, taking a copy of the mail file with her. As long as the user does not modify the same mail file during the trip, the current PersonalRAID design can be trivially extended to accommodate such concurrent but non-conflicting updates.

After the user disconnects the VA, as the office computer receives new updates, it records the updates in a $\mathrm{V}^3\mathrm{A}$ on the local disk (as described in Section 3.7). When the user returns and connects the mobile VA device to the office computer, as long as there is no conflicting update, the system can merge the checkpoint of the VA with that of the $\mathrm{V}^3\mathrm{A}$ to arrive at a consistent VA image as a result of propagating the $\mathrm{V}^3\mathrm{A}$ updates to the VA. After the merging, the system operates as described previously.

If there are conflicting updates, application or user-level intervention is necessary. In this case, a more sophisticated extension to the current PersonalRAID design is necessary. The logical disk approach upon which the PersonalRAID design is based becomes a convenient and powerful vehicle to support file system versioning. At VA disconnection time, the content of the local disk L-to-P map is "frozen" to represent a version (V_0). As updates are recorded in the local $\mathrm{V}^3\mathrm{A}$, none of the old blocks in V_0 are overwritten. The union of V_0 and the $\mathrm{V}^3\mathrm{A}$ represents a new version V_1. After the user returns and connects her VA device, which may contain conflicting updates, the union of V_0 and the VA represents yet another version V_2. The user or an application must resolve conflicts to arrive at a "consistent" new version V_3. Upon conflict resolution, the old versions V_0, V_1, and V_2 can be freed and a consistent VA image again emerges.

3.8.2 Mobile Storage Limitations

Mobile storage technologies are likely to lag behind conventional ones in terms of performance and capacity. We believe that the performance disadvantages are addressed by the log-structured design of the PersonalRAID and the judicious use of the host disk in the forms such as the V^3A described in Section 3.7. Currently, we do not address the capacity constraint. We are currently researching ways that a weak network connection (when one is present) may complement the mobile storage device to address this limitation.

3.8.3 Limitations of the Log-Structured Organization

The potential disadvantages of the log-structured organization are the possible destruction of read locality and the cost of segment cleaning (or disk garbage collection) [13, 14]. PersonalRAID mainly targets personal computing workloads, which are often bursty and leave ample idle time for cleaning. The cleaning overhead can be further reduced by using techniques like freeblock scheduling [7], and by buying bigger disks and keeping the disk utilization low.

The base LLD design that we have borrowed has several potential disadvantages [1]. Keeping the entire logical-to-physical map in memory, the base LLD design consumes a large amount of memory and incurs some latency when reading this map from disk into memory at startup time. It is, however, possible to cache only a portion of the map in memory and demand it in gradually. We have not implemented this possible optimization.

4 Implementation

In this section, we describe a Linux PersonalRAID implementation. As explained in Section 3, our system implements all the PersonalRAID functionalities at the logical disk level. The system consists of two main components: the PR Driver (PRD) and the PR Server (PRS) (shown in Figure 5). The PRD is a pseudo-block device driver that exports the interface of a disk. Upon receiving I/O requests, the PRD forwards them to the PRS via upcalls. The PRS is a user-space process that implements the LLD abstraction; it manages a partition on the local disk and the VA device in a log-structured manner as described in Section 3.

Most of the complexity in our system is concentrated in the PRS. Despite the upcall overhead, which is purely an implementation artifact, we chose this design for ease of programming, debugging and portability. Unfortunately, this decision also leads to some deadlock possibilities. A user process can cause the buffer cache to flush as a side effect of requesting system services (like dynamic memory allocation). The user process blocks till the flush is completed. If the PRS, which is just another user process, causes a flush to the PRD, the system enters a deadlock with the PRD waiting for the PRS to service the flush request. To prevent such deadlocks, our PRS is designed not to allocate memory dynamically during its life time. All memory used is allocated and locked at start-up time. Also, the PRS does raw I/O using the Linux `/dev/raw/rawN` interface, bypassing the buffer cache. Two additional changes are made in the kernel path used by the PRS to ensure that it never causes a PRD flush.

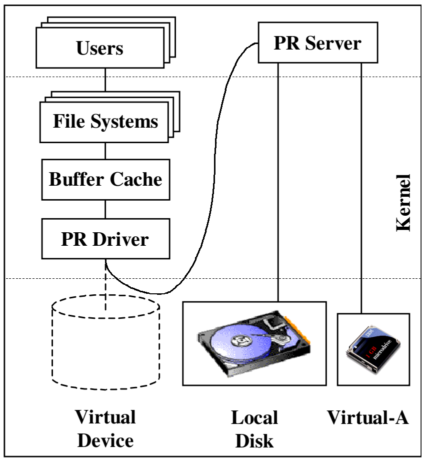

Figure 5: The PR Driver (PRD) and the PR Server (PRS) are the two main components of the PersonalRAID prototype. The PRD is implemented as a dynamically Loadable kernel module, whereas the PRS is a user-space process.

In addition to the in-memory map described in Section 3 (see Figure 4), the PRS maintains several main-memory segments for both the local disk and the VA device. These segments accumulate new block writes, just like the LLD system. Our implementation maintains more than one main-memory segment for each of the two devices. This feature allows us to decouple the amount of write-behind buffering from the segment size, which may need to be based on other considerations such as segment-cleaning performance.

Segment cleaning is begun when the number of clean segments falls below a threshold. We use twice the number of main-memory segments as the threshold. Cleaning is invoked, if necessary, after flushes and stops once the threshold is reached. For simplicity, on the local device, like the LLD system, it chooses the segment with the minimum number of live blocks to clean. Live blocks are read into main memory and are copied with their old timestamps to the main-memory segments. Note that we mix the new block writes with the old blocks.

Segment cleaning for the VA device is a little more complex since replaying is integrated with cleaning. Our current cleaning policy adopts a simple heuristic to strike a balance between two goals: quickly generating empty segments on the VA and quickly propagating data. The cleaner gives higher preference to segments that have data blocks that need to be propagated. Among all such segments, a segment with the minimum number of such blocks is chosen. Among segments with no blocks that need to be propagated, we give preference to segments with fewer live blocks. To clean a VA device segment, live blocks are read into memory and if necessary, propagated to local disk. If the block still needs to be propagated to some other device, it is written back to the VA; otherwise, it is discarded. There is one subtlety in this algorithm. If a VA segment, while being cleaned, contributes new blocks to some main-memory segments of the local disk, then this VA segment cannot be reused until all those local disk segments are safely on the local disk. Otherwise, a crash may lead to a situation where the only copy of a data block is on a remote host, and recovering from such a crash would require visiting that remote host. Thus, before marking a set of segments cleaned in the current cleaning phase as free, we check for this condition and possibly flush the main-memory segments of the local disk.

Currently, the PRS performs request satisfaction, segment flushing, and cleaning sequentially. It is possible that performance can be improved by using multiple threads for some or each of these different tasks. The PRS is about 4000 lines of C code, including a substantial amount of debugging and testing code. The PRD is implemented as a dynamically loadable kernel module. It is about 700 lines of C code. All of the algorithms described above have been implemented with the exception of crash recovery, recovery from device loss, and reconfiguration.

5 Experimental Results

In this section, we evaluate the performance of the prototype PersonalRAID system to demonstrate two conclusions: (1) PersonalRAID can achieve the transparency and reliability goals without imposing a significant performance penalty on a mobile storage user, and (2) the log-structured organization of the PersonalRAID is a sound design choice.

5.1 Experimental Platform

Our experimental setup consists of two laptop end hosts (A & B) with an IBM 1GB Microdrive acting as the VA device. Table 1 shows the configuration of the laptops and Table 2 gives the characteristics of the laptops' internal local disks and the Microdrive. Note that the Microdrive's bandwidth is more than an order of magnitude worse than that of the internal disk. A challenge to the PersonalRAID system is to shield the user from this performance gap.

Model	**Dell Inspiron 4000 Notebook**
Processor	Pentium III, 800 MHz
Memory	256 MB
Operating System	Red Hat Linux 6.2 Kernel 2.2.18

Table 1: Host configuration.

	Local Disk	**VA**
Maker	IBM	IBM
Model	Travelstar 20GN	1 GB Microdrive
Interface	ATA-4	PCMCIA
Capacity (GB)	10	1
RPM	4200	3600
Bandwidth (MB/s)	16.9	1.5
Avg. Latency (ms)	7.1	20.3

Table 2: Characteristics of the local disks and the VA used in the PersonalRAID prototype. The published sustained data rate of the Microdrive is 2.6 MB/s; while our best effort micro-benchmark on the drive yields 1.5 MB/s.

Table 3 shows the configuration parameters of the PR Server. The user "sees" a 2 GB PersonalRAID logical disk, which is larger than the VA capacity. The segment sizes and the numbers of outstanding segments are chosen so that the write-behind buffer of the local partition is smaller than that of the VA: the former is kept small so that the system limits the amount of data loss in case of a crash, while the latter is kept large to allow overwritten data to be deleted before it reaches the Microdrive and mask the higher latency. We must, however, remark that in our experiments, the large size of the VA buffers did not make a significant difference. The reason for this is that a Linux ext2 file system internally uses large buffers to absorb most of the overwrites. Thus, very few overwrites occur in the PersonalRAID buffers, which are at the logical disk level below the file system. In a system where the file system does less buffering, the impact of having large VA buffers might be significant.

5.2 Benchmarks

We report results for two benchmarks. The first is an enhanced version of the "Modified Andrew

	Local Partition	VA
Block Size (KB)	4	4
Seg. Size (MB)	0.5	1
Outstanding Segs	2	8
Size (GB)	3	1

Table 3: Configuration parameters of our 2GB PersonalRAID server.

Benchmark" [4, 9], which we call "MMAB". (We modified the benchmark because the 1990 benchmark does not generate much I/O activity by today's standards.) MMAB has four phases. The first phase creates a directory tree of 50,000 directories, in which every non-leaf directory (with the exception of one) has ten subdirectories. The second phase creates one large file and many small files. The large file created has a size of 256 MB. Each of the small files is 4 KB. The benchmark creates five small files in each of the directories in the first five levels of the directory tree, resulting in a total of about 55,000 small files. The third phase performs file-attribute operations. During this phase, the benchmark first performs a recursive `touch` on all the directories and files in the directory tree; it then computes disk usage of the directory tree by invoking `du`. The fourth and final phase reads files. It first performs a `grep` on each file; it then reads all the files again by performing a `wc` on each file. We run the MMAB benchmark on laptop A and evaluate various options of gaining access to the resulting files on laptop B without the benefit of a network.

The second benchmark is a software development workload. We examine the cost of installing and compiling the Mozilla source code as a user moves between the two laptops. The benchmark has two phases. The first phase creates a development source tree from a compressed archive file (a `.tar.gz` file) stored on a local disk. The source tree consumes about 405 MB of total disk space. We refer to this phase as "MOZ1" in the subsequent sections. During the second phase, we compile the "layout" module within the Mozilla source tree, generating an additional 80 MB of data. We call this phase "MOZ2". To run this benchmark, we start with MOZ1 on laptop A. We then transport the data to the disconnected laptop B to continue with MOZ2.

5.3 Recording Performance

Table 4 details the recording performance of the MMAB benchmark. "UFS-Local" is a Linux "ext2" file system created directly on the local disk partition. "UFS-Upcalls" is the same file system implemented with kernel upcalls into a user-level server. "LLD-Local" uses the same kernel upcall and user-level server mechanisms but it replaces the UFS with a log-structured logical disk organization. The results of these experiments are used to establish the base-line performance, to factor out the cost of using a user-level server, and to quantify the benefit of using an LLD for accessing the local disk. "UFS-MD" is an ext2 file system created directly on the Microdrive. "PR-VA" is an ext2 file system created on the logical disk exported by the PersonalRAID system where the Microdrive is used as the VA device. "PR-VVA" is similar to PR-VA except that it uses a partition on the local disk as the VA device. The PR-VVA performance is an indication of how well the PersonalRAID might perform if the VA device has much better performance than that of the Microdrive.

The Linux ext2 file system performs *both* metadata and data writes asynchronously to the buffer cache. The large memory filters out overwritten data before it reaches the disks and allows the surviving write requests to be intelligently scheduled. As a result, the additional benefit that the log-structured PersonalRAID derives from asynchronous writes is smaller than one might expect. During the MMAB and MOZ experiments, the segment cleaners in the PRS did not get invoked. Later in this section, we describe a separate experiment designed to measure the overhead of segment cleaning.

The results show that the PR-VA system is successful in masking the 10× bandwidth difference between the local disk and the Microdrive: despite the fact that the PersonalRAID needs to write to two devices and that the system incurs the cost of kernel upcalls, the performance of PR-VA is close to or better than that of UFS-Local in most cases due to the relatively low overhead of log-structured recording. An exception is the large write (lwrite) performance, a case where the PersonalRAID recording performance is being limited by the Microdrive bandwidth. The read performance of the PR-VA is also excellent, since unlike the UFS-MD system which performs reads from the slower Microdrive, it satisfies reads from the faster local partition. Finally, the performance of PR-VVA indicates that the PersonalRAID system becomes even more attractive with a faster mobile storage technology.

Table 5 presents the recording performance of the MOZ benchmark, along with the cumulative totals from MMAB. Recall that MOZ1 is the source-unpacking phase run on laptop A and MOZ2 is the compiling phase run on laptop B. The former is more I/O intensive than the latter and it is much more difficult to mask the recording overhead during MOZ1 unless the VA device is faster. After we connect the

	`mkdir` (s)	`lwrite` (s)	`swrite` (s)	`touch` (s)	`du` (s)	`grep` (s)	`wc` (s)	total (s)
UFS-Local	156	20	398	806	6	142	368	1896
UFS-Upcalls	397	33	675	951	15	220	388	2679
LLD-Local	184	28	229	330	5	66	94	936
UFS-MD	398	320	725	1440	1200	535	880	5498
PR-VA	325	239	484	350	167	85	127	1777
PR-VVA	192	55	263	336	151	75	100	1172

Table 4: Detailed breakdown of recording performance for the MMAB benchmark. `mkdir` *is the directory creation phase.* `lwrite` *creates a large file and* `swrite` *creates many small files.* `touch` *and* `du` *perform attribute operations. The* `touch` *phase reads directories and inodes, and writes inodes. The* `du` *phase generates write as well as read traffic because the recursive visit alters access times that are stored in the inodes. The metadata cache misses are the main contributor of this phase's latency.* `grep` *and* `wc` *read all the files.*

	MMAB total (s)	MOZ1 (s)	MOZ2 (s)	
			post-replay	pre-replay
UFS-Local	1896	75	476	—
UFS-Upcalls	2679	126	481	—
LLD-Local	936	66	464	—
UFS-MD	5498	404	640	—
PR-VA	1777	395	548	558
PR-VVA	1172	102	482	486

Table 5: Recording performance.

	Cleaner not invoked (s)	Cleaner invoked (s)
PR-VA	878	928
PR-VVA	713	735

Table 6: Effect of cleaning on recording performance.

	MMAB (s)	MOZ1 (s)	MOZ2 (s)
PR-VA	7.3	7.3	7.3
`tar`	1225	268	—
`hoard`	—	—	62

Table 7: Disconnection performance.

VA device to laptop B, we have two options for PR-VA: we can run MOZ2 immediately after connection, in which case the system reads data from the slower Microdrive; or we can run MOZ2 after replaying the entire content of VA to the local partition, in which case the system reads data from the faster local disk. The workload being CPU-intensive, there is little difference between the 2 cases.

The next set of experiments (see Table 6) capture the effect of invoking the segment cleaner on recording performance. We repeatedly perform a series of recompilation steps where each compilation step is triggered by modifying the file attributes of a small randomly-chosen subset of source files. We perform the experiments on two configurations: a 1 GB VA partition and a 550 MB VA partition. The disk utilization of the first configuration is 50% and the cleaner is not triggered during compilation. The disk utilization of the second configuration is 96% and the cleaner must run to continuously generate clean segments to accommodate the new data generated by the compilation. A total of 231 MB of data is generated during the experiment. The cost attributed to the cleaner is low.

5.4 Disconnection and Connection Performance

Table 7 and 8 compare the disconnection and connection latencies of several alternatives. To disconnect/connect the VA device from/to a host in PersonalRAID, all it takes is writing/reading the VA checkpoint. The checkpoint write may be preceded by the flushing of the remaining memory segments, which may add a few more seconds. The benchmarks that we used sync the disk at the end of each benchmark run so both the checkpoint write and read times are constants.

We examine two simpler alternatives to PersonalRAID. One is to use the Unix `tar` utility to create/unpack a Unix archive on the Microdrive at disconnection/connection times. `tar` writes less data to the Microdrive than PersonalRAID because it writes only enough information to allow it to recreate the directory structure without physically copying all the blocks, eliminating fragmentation costs. Unpacking time is much faster than packing time because unpacking is benefitting from the asynchronous writes of the Linux ext2 file system. Despite these optimizations, the latencies are not tolerable.

For MOZ2, `tar` alone would not have been adequate because we need to identify the files that have been changed, added, or deleted after compilation and act on just the changes. For this purpose, we use a pair of Unix scripts `hoard/unhoard`. Although these scripts are admittedly crude due to the liberal forking of some Unix processes, it is clear from the data that the resulting latencies of this approach are

	MMAB (s)	MOZ1 (s)	MOZ2 (s)
PR-VA	7.0	7.0	7.0
`untar`	411	225	—
`unhoard`	—	—	67

Table 8: Connection performance.

	MMAB (s)	MOZ1 (s)	MOZ2 (s)
VA→LFS	553	340	56
VA→UFS	988	547	100
VVA→LFS	159	90	17

Table 9: Replaying performance.

unlikely to be satisfactory.

To be fair, we note that the connection times for the simpler alternatives described above include the replay time. These times are less than the sum of connection and replay times for the PersonalRAID system (see the top rows in Tables 8 and 9), because the simpler alternatives write less data to the Microdrive than PersonalRAID, and write their data as a large, sequential file. Unlike PersonalRAID, however, these simpler alternatives do not allow normal operations to overlap the replay time and thus have a much greater impact on the user.

5.5 Replaying Performance

After a VA device is connected to a PersonalRAID host, background propagation (or replaying) starts. Table 9 shows the results of the experiments designed to analyze the impact of the log-structured organization on the replaying performance. "VA→LFS" refers to the PersonalRAID that replays from a log-structured VA to a log-structured local partition.

An alternative to this design is to use a UFS-style update-in-place organization on the local disk partition, while retaining the log-structured organization on the VA so that reads for replaying still occur at segment-sized granularity. To realize this alternative design, all we need to change is the L-to-P mapping for the local disk; we substitute it with an "identity mapping", which simply maps a logical address received from the file system to an identical physical address. The PR Server employs helper threads to perform asynchronous writes to the local partition and it limits the maximum number of outstanding writes to 20. Table 9 refers to this alternative as "VA→UFS".

Finally, to evaluate the impact of a faster mobile storage device, we replay from a local partition-based VVA (as described in Section 3.7) to another local partition, both of which are log-structured. Table 9 refers to this last alternative as "VVA→LFS".

During these experiments, the cleaner on the log-structured local partition did not get invoked. Both cleaning and replaying are background activities that can result in synergistic benefits. We plan to investigate the performance impact of this integration in the near future. Furthermore, since all the storage devices in a PersonalRAID are self-contained, it is possible for end hosts to independently clean their local disks when they are not in use. Therefore, it is possible that large number of free segments are available before replaying starts.

Table 9 shows that despite the fact that both the VA→LFS and the VA→UFS configurations are limited by the slow read performance of the Microdrive, the former can replay significantly faster than the latter, thanks in no small part to the former's log-structured organization. If the VA device could perform reads faster, the impact of the log-structured organization would have been even more dramatic as implied by the VVA→LFS performance numbers.

6 Related Work

Although PersonalRAID can be extended to deal with conflicting updates (as discussed in Section 3.8.1) and this is one of our ongoing research topics, the primary use cases targeted by PersonalRAID today are just that: personal usage scenarios where the availability of a single coherent name space and reliability are the primary concerns but conflict resolution is not. Conflicts are inherently file system-level or application-level events that must be addressed at these higher levels. PersonalRAID is a storage-level solution that can only provide mechanisms such as versioning that higher level systems may exploit. A number of research systems (including Ficus [11], Coda [6], and Bayou [10, 15]) have focused on conflict resolution techniques that may provide insight on how to build and extend services running on top of PersonalRAID.

Many systems share PersonalRAID's goal of synchronizing the contents of a number of hosts; these systems include disconnected client/server systems such as Coda [6], distributed applications such as those built on top of Bayou [10, 15], and replicated databases such as those provided by Oracle [8] and Sybase [2]. A common technique is to use an operations log that is recorded at the site that initiates the updates and is replayed at the various replicas. A variation of the theme is the use of "asynchronous RPCs" as those employed by Rover [5]. Before log replaying is complete, the access to a replica needs to be suspended if one does not want to expose stale data. We have determined that neither stale data nor the latency involved in log propagation may be tolerable for a PersonalRAID user. Thanks to its LFS roots, PersonalRAID inherits the absence of the notion of a separate operations log—the collection of distributed logs *is* the storage system. As a result, the fresh updates carried in the VA device are always immediately accessible while replaying can occur in the background when convenient.

A VA device is similar to a disconnected Coda client in that it "hoards" data and buffers the latest changes [6]. It is different in that the VA device is not meant to support I/O operations on its own: the role of the VA in a PersonalRAID is two fold: (1) it acts as a transporter of all updates that are used to synchronize the contents of several disconnected end hosts, some of which may be mobile; and (2) it supports I/O operations only when it is coupled with a host local disk. There is no user involvement or guesswork involved in determining the content of the VA; and there is no danger of a "hoard miss."

The use of a mobile device to carry updates to other weakly connected hosts to bring about eventual consistency via pair-wise communications is similar to the approach taken by Bayou [10, 15]. Bayou provides a framework for application-specific conflict resolution and applications must be reprogrammed or developed from scratch to take advantage of the Bayou infrastructure. PersonalRAID, as a storage system, does not resolve conflicts in itself; as a result, it is possible for us to develop a general system, on top of which existing personal applications may run unmodified.

Existing mobile systems typically do not address data reliability on mobile hosts: these mobile hosts are typically considered inferior "second-class citizens" and their data is vulnerable until they are propagated to "first-class citizen" servers that are professionally managed and backed up occasionally. PersonalRAID provides protection against any single device loss at all times by leveraging the exact same mechanism that is needed to bring about eventual consistency of the system.

Finally, there are existing applications such as the Windows "Briefcase" that can synchronize the contents of multiple hosts. The two problems with these applications are exactly the problems that PersonalRAID is designed to address: (1) the inconvenience involved in manual movement of data, and (2) the poor performance in terms of both latency and throughput during synchronization events.

7 Conclusion

As storage technology advances, a user is facing an increasing array of disconnected storage devices. Two of the important challenges a user must face is the lack of a single transparent storage space that is ubiquitously available and a certain degree of reliability assurance. PersonalRAID is a mobile storage management system that attacks these two problems. At the heart of a PersonalRAID is a mobile Virtual-A device that allows the user to transparently transport, replicate, and access data while interacting with a number of disconnected storage devices. By employing a distributed log-structured organization, the system is able to accomplish these tasks without imposing any serious performance penalty on the user.

Acknowledgments

We would like to thank Brent Waters, Victor Shnayder and Atul Pokharel for their ideas and help during the initial part of the project. Thanks are also due to our shepherd David Kotz and other FAST reviewers for their detailed and critical comments.

References

[1] de Jonge, W., Kaashoek, M. F., and Hsieh, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 15–28.

[2] Gorelik, A., Wang, Y., and Deppe, M. Sybase Replication Server. In *Proc. ACM SIGMOD Conference* (May 1994), p. 468.

[3] Growchowski, E. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. In *Datatech* (September 1998), ICG Publishing, pp. 11–16.

[4] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[5] Joseph, A. D., deLespinasse, A. F., Tauber, J. A., Gifford, D. K., and Kaashoek, M. F. Rover: A Toolkit for Mobile Information Access. In *Proc. the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 156–171.

[6] Kistler, J., and Satyanarayanan, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[7] Lumb, C., Schindler, J., Ganger, G. R., Riedel, E., and Nagle, D. F. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth from Busy Disk Drives. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000).

[8] Oracle Corporation. *Oracle7 Server Distributed Systems: Replicated Data*, 1994.

[9] Ousterhout, J. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proc. of the 1990 Summer USENIX* (June 1990).

[10] Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. the 16th ACM Symposium on Operating Systems Principles* (October 1997), pp. 288–301.

[11] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. Resolving File Conflicts in the Ficus File System. In *Proc. the Summer USENIX Conference* (June 1994), pp. 183–195.

[12] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[13] Seltzer, M., Bostic, K., McKusick, M., and Staelin, C. An Implementation of a Log-Structured File System for UNIX. In *Proc. of the 1993 Winter USENIX* (Jan. 1993), pp. 307–326.

[14] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., and Padmanabhan, V. File System Logging Versus Clustering: A Performance Comparison. In *Proc. of the 1995 Winter USENIX* (Jan. 1995).

[15] Terry, D. B., Theimer, M. M., Peterson, K., Demers, A. J., Spreitzer, M. J., and Hauaser, C. H. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 172–183.