Integrating Portable and Distributed Storage

Niraj Tolia[†‡], Jan Harkes[†], Michael Kozuch[‡], and M. Satyanarayanan[†‡]
[†]Carnegie Mellon University and [‡]Intel Research Pittsburgh

Abstract

We describe a technique called *lookaside caching* that combines the strengths of distributed file systems and portable storage devices, while negating their weaknesses. In spite of its simplicity, this technique proves to be powerful and versatile. By unifying distributed storage and portable storage into a single abstraction, lookaside caching allows users to treat devices they carry as merely performance and availability assists for distant file servers. Careless use of portable storage has no catastrophic consequences. Experimental results show that significant performance improvements are possible even in the presence of stale data on the portable device.

1 Introduction

Floppy disks were the sole means of sharing data across users and computers in the early days of personal computing. Although they were trivial to use, considerable discipline and foresight was required of users to ensure data consistency and availability, and to avoid data loss — if you did not have the right floppy at the right place and time, you were in trouble! These limitations were overcome by the emergence of distributed file systems such as NFS [24], Netware [8], LanManager [34], and AFS [7]. In such a system, responsibility for data management is delegated to the distributed file system and its operational staff.

Personal storage has come full circle in the recent past. There has been explosive growth in the availability of USB- and Firewire-connected storage devices such as flash memory keychains and portable disk drives. Although very different from floppy disks in capacity, data transfer rate, form factor, and longevity, their usage model is no different. In other words, they are just glorified floppy disks and suffer from the same limitations mentioned above. Why then are portable storage devices in such demand today? Is there a way to use them that avoids the messy mistakes of the past, where a user was often awash in floppy disks trying to figure out which one had the latest version of a specific file? If loss, theft or destruction of a portable storage device occurs, how can one prevent catastrophic data loss? Since human attention grows ever more scarce, can we reduce the data management demands on attention and discipline in the use of portable devices?

We focus on these and related questions in this paper. We describe a technique called *lookaside caching* that combines the strengths of distributed file systems and portable storage devices, while negating their weaknesses. In spite of its simplicity, this technique proves to be powerful and versatile. By unifying "storage in the cloud" (distributed storage) and "storage in the hand" (portable storage) into a single abstraction, lookaside caching allows users to treat devices they carry as merely performance and availability assists for distant file servers. Careless use of portable storage has no catastrophic consequences.

We begin in Section 2 by examining the strengths and weaknesses of portable storage and distributed file systems. We describe the design of lookaside caching in Section 3 followed by a discussion of related work in Section 4. Section 5 describes the implementation of lookaside caching. We quantify the performance benefit of lookaside caching in Section 6, using three different benchmarks. We explore broader use of lookaside caching in Section 7, and conclude in Section 8 with a summary.

2 Background

To understand the continuing popularity of portable storage, it is useful to review the strengths and weaknesses of portable storage and distributed file systems. While there is considerable variation in the designs of distributed file systems, there is also a substantial degree of commonality across them. Our discussion below focuses on these common themes.

Performance: A portable storage device offers uniform performance at all locations, independent of factors such as network connectivity, initial cache state, and temporal locality of references. Except for a few devices such as floppy disks, the access times and bandwidths of portable devices are comparable to those of local disks. In contrast, the performance of a distributed file system is highly variable. With a warm client cache and good locality, performance can match local storage. With a cold cache, poor connectivity and low locality, performance can be intolerably slow.

Availability: If you have a portable storage device in hand, you can access its data. Short of device failure, which is very rare, no other common failures prevent data access. In contrast, distributed file systems are susceptible to network failure, server failure, and a wide range of operator errors.

Robustness: A portable storage device can easily be lost, stolen or damaged. Data on the device becomes permanently inaccessible after such an event. In contrast, data in

a distributed file system continues to be accessible even if a particular client that uses it is lost, stolen or damaged. For added robustness, the operational staff of a distributed file system perform regular backups and typically keep some of the backups off site to allow recovery after catastrophic site failure. Backups also help recovery from user error: if a user accidentally deletes a critical file, he can recover a backed-up version of it. In principle, a highly disciplined user could implement a careful regimen of backup of portable storage to improve robustness. In practice, few users are sufficiently disciplined and well-organized to do this. It is much simpler for professional staff to regularly back up a few file servers, thus benefiting all users.

Sharing/Collaboration: The existence of a common name space simplifies sharing of data and collaboration between the users of a distributed file system. This is much harder if done by physical transfers of devices. If one is restricted to sharing through physical devices, a system such as PersonalRAID [26] can be valuable in managing complexity.

Consistency: Without explicit user effort, a distributed file system presents the latest version of a file when it is accessed. In contrast, a portable device has to be explicitly kept up to date. When multiple users can update a file, it is easy to get into situations where a portable device has stale data without its owner being aware of this fact.

Capacity: Any portable storage device has finite capacity. In contrast, the client of a distributed file system can access virtually unlimited amounts of data spread across multiple file servers. Since local storage on the client is merely a cache of server data, its size only limits working set size rather than total data size.

Security: The privacy and integrity of data on portable storage devices relies primarily on physical security. A further level of safety can be provided by encrypting the data on the device, and by requiring a password to mount it. These can be valuable as a second layer of defense in case physical security fails. Denial of service is impossible if a user has a portable storage device in hand. In contrast, the security of data in a distributed file system is based on more fragile assumptions. Denial of service may be possible through network attacks. Privacy depends on encryption of network traffic. Fine-grain protection of data through mechanisms such as access control lists is possible, but relies on secure authentication using a mechanism such as Kerberos [28].

Ubiquity: A distributed file system requires operating system support. In addition, it may require environmental support such as Kerberos authentication and specific firewall configuration. Unless a user is at a client that meets all of these requirements, he cannot access his data in a distributed file system. In contrast, portable storage only depends on widely-supported low-level hardware and software interfaces. If a user sits down at a random machine, he can be much more confident of accessing data from portable storage in his possession than from a remote file server.

3 Lookaside Caching

Our goal is to exploit the performance and availability advantages of portable storage to improve these same attributes in a distributed file system. The resulting design should preserve all other characteristics of the underlying distributed file system. In particular, there should be no compromise of robustness, consistency or security. There should also be no added complexity in sharing and collaboration. Finally, the design should be tolerant of human error: improper use of the portable storage device (such as using the wrong device or forgetting to copy the latest version of a file to it) should not hurt correctness.

Lookaside caching is an extension of AFS2-style whole-file caching [7] that meets the above goals. It is based on the observation that virtually all distributed file system protocols provide separate remote procedure calls (RPCs) for access of meta-data and access of data content. Lookaside caching extends the definition of meta-data to include a cryptographic hash of data content. This extension only increases the size of meta-data by a modest amount: just 20 bytes if SHA-1 [15] is used as the hash. Since hash size does not depend on file length, it costs very little to obtain and cache hash information even for many large files. Using POSIX terminology, caching the results of "`ls -lR`" of a large tree is feasible on a small client, even if there is not enough cache space for the contents of all the files in the tree. This continues to be true even if one augments `stat` information for each file or directory in the tree with its SHA-1 hash.

Once a client possesses valid meta-data for an object, it can use the hash to redirect the fetch of data content. If a mounted portable storage device has a file with matching length and hash, the client can obtain the contents of the file from the device rather than from the file server. Whether it is beneficial to do this depends on factors such as file size, network bandwidth, and device transfer rate. The important point is that possession of the hash gives a degree of freedom that clients of a distributed file system do not possess today.

Since lookaside caching treats the hash as part of the meta-data, there is no compromise in consistency. The underlying cache coherence protocol of the distributed file system determines how closely client state tracks server state. There is no degradation in the accuracy of this tracking if the hash is used to redirect access of data content. To ensure no compromise in security, the file server should return a null hash for any object on which the client only has permission to read the meta-data.

Lookaside caching can be viewed as a degenerate case of the use of *file recipes,* as described by Tolia *et al.* [31]. In that work, a recipe is an XML description of file content that enables block-level reassembly of the file from content-addressable storage. One can view the hash of a file as the smallest possible recipe for it. The implementation using recipes is considerably more complex than our support for lookaside caching. In return for this complexity, synthesis

from recipes may succeed in many situations where lookaside fails.

4 Related Work

Lookaside caching has very different goals and design philosophy from systems such as PersonalRAID [26], Segank [25], and Footloose [18]. Our starting point is the well-entrenched base of distributed file systems in existence today. We assume that these are successful because they offer genuine value to their users. Hence, our goal is to integrate portable storage devices into such a system in a manner that is minimally disruptive of its existing usage model. In addition, we make no changes to the native file system format of a portable storage device; all we require is that the device be mountable as a local file system at any client of the distributed file system. In contrast, all the above systems takes a much richer view of the role of portable storage devices. They view them as first-class citizens rather than as adjuncts to a distributed file system. They also use customized storage layouts on the devices. Therefore, our design and implementation are much simpler, but also more limited in functionality.

Another project with overlapping goals is the Personal Server [32] effort. This system tries to integrate computation, communication, and storage to provide ubiquitous access to personal information and applications. However, being a mobile computer, it is more heavyweight in terms of the hardware requirements. There are also a number of commercial solutions providing mobility solutions through the use of portable storage devices. Migo [12], one of these products, has combined a USB portable storage device with synchronization software for personal files, email, and other settings. However, these solutions focus exclusively on the use of the portable device and do not integrate network storage.

The use of cryptographic hashes to describe data has been explored earlier in a variety of different contexts. Spring *et al.* [27] used the technique to identify and remove redundant network traffic. The Single Instance Storage [3] and the Venti [20] systems use cryptographic hashes to remove duplicate content at the file and block level respectively. Unlike lookaside caching, a number of other systems such as CASPER [31] and LBFS [14] prefer to further subdivide objects. This slightly more complicated approach usually uses an algorithm similar to the Rabin fingerprinting technique [10, 21]. For lookaside caching, it was a conscious decision to favor the simplest possible design. It is also well known that the use of hashes can leak information. In the context of lookaside caching, fetching a SHA-1 hash without fetching the corresponding contents can indicate that the client already possessed the data. As shown by Mogul *et al.* [13], this can allow a malicious server to inspect a client's cache. The most obvious solution is to only allow lookaside caching with trusted servers. As we believe that the predominant use of lookaside caching will be with trusted servers, this solution should not significantly impact users.

5 Prototype Implementation

We have implemented lookaside caching in the Coda file system on Linux. The user-level implementation of Coda client cache manager and server code greatly simplified our effort since no kernel changes were needed. The implementation consists of four parts: a small change to the client-server protocol; a quick index check (the "lookaside") in the code path for handling a cache miss; a tool for generating lookaside indexes; and a set of user commands to include or exclude specific lookaside devices.

The protocol change replaces two RPCs, `ViceGetAttr()` and `ViceValidateAttrs()` with the extended calls `ViceGetAttrPlusSHA()` and `ViceValidateAttrsPlusSHA()` that have an extra parameter for the SHA-1 hash of the file. `ViceGetAttr()` is used to obtain meta-data for a file or directory, while `ViceValidateAttrs()` is used to revalidate cached meta-data for a collection of files or directories when connectivity is restored to a server. Our implementation preserves compatibility with legacy servers. If a client connects to a server that has not been upgraded to support lookaside caching, it falls back to using the original RPCs mentioned above.

The lookaside occurs just before the execution of the `ViceFetch()` RPC to fetch file contents. Before network communication is attempted, the client consults one or more lookaside indexes to see if a local file with identical SHA-1 value exists. Trusting in the collision resistance of SHA-1 [11], a copy operation on the local file can then be a substitute for the RPC. To detect version skew between the local file and its index, the SHA-1 hash of the local file is recomputed. In case of a mismatch, the local file substitution is suppressed and the cache miss is serviced by contacting the file server. Coda's consistency model is not compromised, although some small amount amount of work is wasted on the lookaside path.

The index generation tool walks the file tree rooted at a specified pathname. It computes the SHA-1 hash of each file and enters the filename-hash pair into the index file, which is similar to a Berkeley DB database [17]. The tool is flexible regarding the location of the tree being indexed: it can be local, on a mounted storage device, or even on a nearby NFS or Samba server. For a removable device such as a USB storage keychain or a DVD, the index is typically located right on the device. This yields a self-describing storage device that can be used anywhere. Note that an index captures the values in a tree at one point in time. No attempt is made to track updates made to the tree after the index is created. The tool must be re-run to reflect those updates. Thus, a lookaside index is best viewed as a collection of *hints* [30].

`cfs lka --clear`	*exclude all indexes*
`cfs lka +db1`	*include index db1*
`cfs lka -db1`	*exclude index db1*
`cfs lka --list`	*print lookaside statistics*

Figure 1. Lookaside Commands on Client

Dynamic inclusion or exclusion of lookaside devices is done through user-level commands. Figure 1 lists the relevant commands on a client. Note that multiple lookaside devices can be in use at the same time. The devices are searched in order of inclusion.

As mentioned earlier, the fact that our system does not modify the portable device's storage layout allows it to use any device that exports a generic file system interface. This allows files to be stored on the device in any manner chosen by the user, including the same tree structure as the distributed file system. For example, in the Kernel Compile benchmark described in Section 6.1, the portable device was populated by simply unarchiving a normal kernel source tree. The advantage of this is that user can still have access to the files in the absence of a network or even a distributed file system client. However, this also allows the user to edit files without the knowledge of the lookaside caching system. While recomputation of the file's hash at the time of use can expose the update, it is up to the user to manually copy the changes back into the distributed file system.

6 Evaluation

How much of a performance win can lookaside caching provide? The answer clearly depends on the workload, on network quality, and on the overlap between data on the lookaside device and data accessed from the distributed file system. To obtain a quantitative understanding of this relationship, we have conducted controlled experiments using three different benchmarks: a kernel compile benchmark, a virtual machine migration benchmark, and single-user trace replay benchmark. The rest of this section presents our benchmarks, experimental setups, and results.

6.1 Kernel Compile

6.1.1 Benchmark Description

Our first benchmark models a nomadic software developer who does work at different locations such as his home, his office, and a satellite office. Network connection quality to his file server may vary across these locations. The developer carries a version of his source code on a lookaside device. This version may have some stale files because of server updates by other members of the development team.

We use version 2.4 of the Linux kernel as the source tree in our benchmark. Figure 2 shows the measured degree of commonality across five different minor versions of the 2.4 kernel, obtained from the FTP site `ftp.kernel.org`. This

Kernel Version	Size (MB)	Files Same	Bytes Same	Release Date	Days Stale
2.4.18	118.0	100%	100%	02/25/02	0
2.4.17	116.2	90%	79%	12/21/01	66
2.4.13	112.6	74%	52%	10/23/01	125
2.4.9	108.0	53%	30%	08/16/01	193
2.4.0	95.3	28%	13%	01/04/01	417

This table shows key characteristics of the Linux kernel versions used in our compilation benchmark. In our experiments, the kernel being compiled was always version 2.4.18. The kernel on the lookaside device varied across the versions listed above. The second column gives the size of the source tree of a version. The third column shows what fraction of the files in that version remain the same in version 2.4.18. The number of bytes in those files, relative to total release size, is given in the fourth column. The last column gives the difference between the release date of a version and the release date of version 2.4.18.

Figure 2. Linux Kernel Source Trees

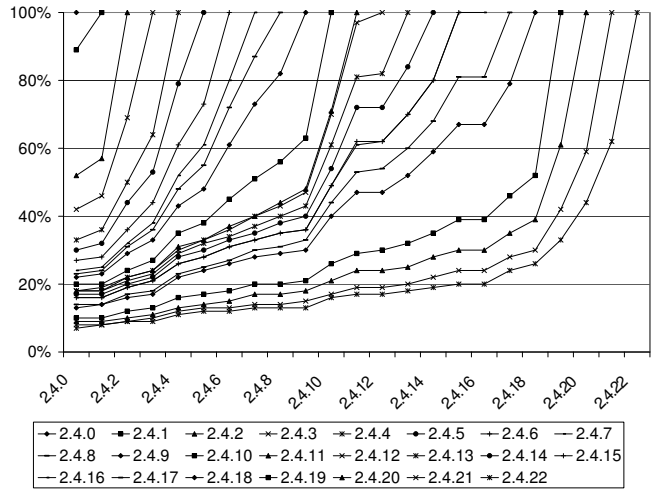
data shows that there is a substantial degree of commonality even across releases that are many weeks apart. Our experiments only use five versions of Linux, but Figure 3 confirms that commonality across minor versions exists for all of Linux 2.4. Although we do not show the corresponding figure, we have also confirmed the existence of substantial commonality across Linux 2.2 versions.

6.1.2 Experimental Setup

Figure 4 shows the experimental setup we used for our evaluation. The client contained a 3.0 GHz Pentium® 4 processor (with Hyper-Threading) with 2 GB of SDRAM. The file server contained a 2.0 GHz Pentium® 4 processor (without Hyper-Threading) with 1 GB of SDRAM. Both machines ran Red Hat 9.0 Linux and Coda 6.0.2, and were connected by 100 Mb/s Ethernet. The client file cache size was large enough to prevent eviction during the experiments, and the client was operated in write-disconnected mode. We ensured that the client file cache was always cold at the start of an experiment. To discount the effect of a cold I/O buffer cache on the server, a warming run was done prior to each set of experiments.

All experiments were run at four different bandwidth settings: 100 Mb/s, 10 Mb/s, 1 Mb/s and 100 Kb/s. We used a NISTNet network router [16] to control bandwidth. The router is simply a standard PC with two network interfaces running Red Hat 7.2 Linux and release 2.0.12 of the NISTNet software. No extra latency was added at 100 Mb/s and 10 Mb/s. For 1 Mb/s and 100 Kb/s, we configured NISTNet to add round trip latencies of 10 ms and 100 ms respectively.

The lookaside device used in our experiments was a 512 MB Hi-Speed USB flash memory keychain. The manufacturer of this device quotes a nominal read bandwidth of 48 Mb/s, and a nominal write bandwidth of 36 Mb/s. We conducted a set of tests on our client to verify these figures. Figure 6 presents our results. For all file sizes ranging from



Each curve above corresponds to one minor version of the Linux 2.4 kernel. That curve represents the measured commonality between the minor version and all previous minor versions. The horizontal axis shows the set of possible minor versions. The vertical axis shows the percentage of data in common. The rightmost point on each curve corresponds to 100% because each minor version overlaps 100% with itself.

Figure 3. Commonality Across Linux 2.4 Versions

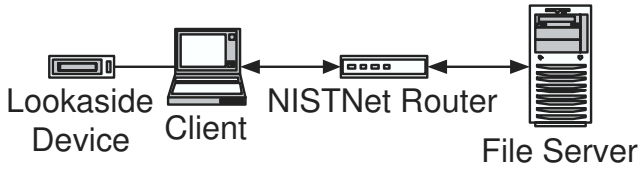


Figure 4. Experimental Setup

4 KB to 100 MB, the measured read and write bandwidths were much lower than the manufacturer's figures.

6.1.3 Results

The performance metric in this benchmark is the elapsed time to compile the 2.4.18 kernel. This directly corresponds to the performance perceived by our hypothetical software developer. Although the kernel being compiled was always version 2.4.18 in our experiments, we varied the contents of the portable storage device to explore the effects of using stale lookaside data. The portable storage device was unmounted between each experiment run to discount the effect of the buffer cache.

Figure 5 presents our results. For each portable device state shown in that figure, the corresponding "Files Same" and "Bytes Same" columns of Figure 2 bound the usefulness of lookaside caching. The "Days Stale" column indicates the staleness of device state relative to the kernel being compiled.

At the lowest bandwidth (100 Kb/s), the win due to lookaside caching is impressive: over 90% with an up-to-

File Size	Measured Data Rate	
	Read (Mb/s)	Write (Mb/s)
4 KB	6.3	7.4
16 KB	6.3	12.5
64 KB	16.7	25.0
256 KB	25.0	22.2
1 MB	28.6	25.8
10 MB	29.3	26.4
100 MB	29.4	26.5

This tables displays the measured read and write bandwidths for different file sizes on the portable storage device used in our experiments. To discount caching effects, we unmounted and remounted the device before each trial. For the same reason, all writes were performed in synchronous mode. Every data point is the mean of three trials; the standard deviation observed was negligible.

Figure 6. Portable Storage Device Performance

date device (improving from 9348.8 seconds to 884.9 seconds), and a non-trivial 10.6% (from 9348.8 seconds to 8356.7 seconds) with data that is over a year old (version 2.4.0)! Data that is over two months old (version 2.4.17) is still able to give a win of 67.8% (from 9348.8 seconds to 3011.2 seconds).

At a bandwidth of 1 Mb/s, the wins are still impressive. They range from 63% (from 1148.3 seconds to 424.8 seconds) with an up-to-date portable device, down to 4.7% (1148.3 seconds to 1094.3 seconds) with the oldest device state. A device that is stale by one version (2.4.17) still gives a win of 52.7% (1148.3 seconds to 543.6 seconds).

On a slow LAN (10 Mb/s), lookaside caching continues to give a strong win if the portable device has current data: 27.1% (388.4 seconds to 282.9 seconds). The win drops to 6.1% (388.4 seconds to 364.8 seconds) when the portable device is one version old (2.4.17). When the version is older than 2.4.17, the cost of failed lookasides exceeds the benefits of successful ones. This yields an overall loss rather than a win (represented as a negative win in Figure 5). The worst loss at 10 Mb/s is 8.4% (388.4 seconds to 421.1 seconds).

Only on a fast LAN (100 Mb/s) does the overhead of lookaside caching exceed its benefit for all device states. The loss ranges from a trivial 1.7% (287.7 seconds to 292.7 seconds) with current device state to a substantial 24.5% (287.7 seconds to 358.1 seconds) with the oldest device state. Since the client cache manager already monitors bandwidth to servers, it would be simple to suppress lookaside at high bandwidths. Although we have not yet implemented this simple change, we are confident that it can result in a system that almost never loses due to lookaside caching.

6.2 Internet Suspend/Resume

6.2.1 Benchmark Description

Our second benchmark is based on the application that forced us to rethink the relationship between portable storage and distributed file systems. *Internet Suspend/Resume*

Bandwidth	Lookaside Device State					
	No Device	2.4.18	2.4.17	2.4.13	2.4.9	2.4.0
100 Mb/s	287.7 (5.6)	292.7 (6.4) [-1.7%]	324.7 (16.4) [-12.9%]	346.4 (6.9) [-20.4%]	362.7 (3.4) [-26.1%]	358.1 (7.7) [-24.5%]
10 Mb/s	388.4 (12.9)	282.9 (8.3) [27.1%]	364.8 (12.4) [6.1%]	402.7 (2.3) [-3.7%]	410.9 (2.1) [-5.8%]	421.1 (12.8) [-8.4%]
1 Mb/s	1148.3 (6.9)	424.8 (3.1) [63.0%]	543.6 (11.5) [52.7%]	835.8 (3.7) [27.2%]	1012.4 (12.0) [11.8%]	1094.3 (5.4) [4.7%]
100 Kb/s	9348.8 (84.3)	884.9 (12.0) [90.5%]	3011.2 (167.6) [67.8%]	5824.0 (221.6) [37.7%]	7616.0 (130.0) [18.5%]	8356.7 (226.9) [10.6%]

These results show the time (in seconds) taken to compile the Linux 2.4.18 kernel. The column labeled "No Device" shows the time taken for the compile when no portable device was present and all data had to be fetched over the network. The column labeled "2.4.18" shows the results when all of the required data was present on the storage device and only meta-data (i.e. `stat` information) was fetched across the network. The rest of the columns show the cases where the lookaside device had versions of the Linux kernel older than 2.4.18. Each data point is the mean of three trials; standard deviations are in parentheses. The numbers in square brackets give the "win" for each case: that is, the percentage improvement over the "no device" case.

Figure 5. Time for Compiling Linux Kernel 2.4.18

(ISR) is a thick-client mechanism that allows a user to suspend work on one machine, travel to another location, and resume work on another machine there [9]. The user-visible state at resume is exactly what it was at suspend. ISR is implemented by layering a virtual machine (VM) on a distributed file system. The ISR prototype layers VMware on Coda, and represents VM state as a tree of 256 KB files.

A key ISR challenge is large VM state, typically many tens of GB. When a user resumes on a machine with a cold file cache, misses on the 256 KB files can result in significant performance degradation. This overhead can be substantial at resume sites with poor connectivity to the file server that holds VM state. If a user is willing to carry a portable storage device with him, part of the VM state can be copied to the device at suspend. Lookaside caching can then reduce the performance overhead of cache misses at the resume site. It might not always be possible to carry the entire VM state as writing it to the portable device may take too long for a user in a hurry to leave. In contrast, propagating updates to a file server can continue after the user leaves.

A different use of lookaside caching for ISR is based on the observation that there is often substantial commonality in VM state across users. For example, the installed code for applications such as Microsoft Office is likely to be the same for all users running the identical software release of those applications [3]. Since this code does not change until a software upgrade (typically many months apart), it would be simple to distribute copies of the relevant 256 KB files on DVD or CD-ROM media at likely resume sites.

Notice that lookaside caching is tolerant of human error in both of the above contexts. If the user inserts the wrong USB storage keychain into his machine at resume, stale data on it will be ignored. Similarly, use of the wrong DVD or CD-ROM does not hurt correctness. In both cases, the user sees slower performance but is otherwise unaffected.

Since ISR is intended for interactive workloads typical of laptop environments, we created a benchmark called the *Common Desktop Application (CDA)* that models an interactive Windows user. CDA uses Visual Basic scripting to drive Microsoft Office applications such as Word, Excel, Powerpoint, Access, and Internet Explorer. It consists of a total of 113 independently-timed operations such as `find-and-replace`, `open-document`, and `save-as-html`. Note that each of these macro-operations may result in more than one file system call within the VM and, consequently, multiple requests to Coda. Minor user-level actions such as keystrokes, object selection, or mouse-clicks are not timed.

6.2.2 Experimental Setup

Our experimental infrastructure consists of clients with 2.0 GHz Pentium® 4 processors connected to a server with a 1.2 GHz Pentium® III Xeon™ processor through 100 Mb/s Ethernet. All machines have 1 GB of RAM, and run Red Hat 7.3 Linux. Unless indicated otherwise, a Hi-Speed USB flash memory keychain was used. Clients use VMware Workstation 3.1 and have an 8 GB Coda file cache. The VM is configured to have 256 MB of RAM and 4 GB of disk, and runs Windows XP as the guest OS. As in the previous benchmark, we use the NISTNet network emulator

	No Lookaside	With Lookaside	Win
100 Mb/s	14 (0.5)	13 (2.2)	7.1%
10 Mb/s	39 (0.4)	12 (0.5)	69.2%
1 Mb/s	317 (0.3)	12 (0.3)	96.2%
100 Kb/s	4301 (0.6)	12 (0.1)	99.7%

This table shows the resume latency (in seconds) for the CDA benchmark at different bandwidths, with and without lookaside to a USB flash memory keychain. Each data point is the mean of three trials; standard deviations are in parentheses.

Figure 7. Resume Latency

to control bandwidth.

6.2.3 Results

From a user's perspective, the key performance metrics of ISR can be characterized by two questions:

- *How slow is the resume step?*
  This speed is determined by the time to fetch and decompress the physical memory image of the VM that was saved at suspend. This is the smallest part of total VM state that must be present to begin execution. The rest of the state can be demand-fetched after execution resumes. We refer to the delay between the resume command and the earliest possible user interaction as *Resume Latency.*
- *After resume, how much is work slowed?*
  The user may suffer performance delays after resume due to file cache misses triggered by his VM interactions. The metric we use to reflect the user's experience is the total time to perform all the operations in the CDA benchmark (this excludes user think time). We refer to this metric as *Total Operation Latency.*

Portable storage can improve both resume latency and total operation latency. Figure 7 presents our results for the case where a USB flash memory keychain is updated at suspend with the minimal state needed for resume. This is a single 41 MB file corresponding to the compressed physical memory image of the suspended virtual machine. Comparing the second and third columns of this figure, we see that the effect of lookaside caching is noticeable below 100 Mb/s, and is dramatic at 100 Kb/s. A resume time of just 12 seconds rather than 317 seconds (at 1 Mb/s) or 4301 seconds (at 100 Kb/s) can make a world of a difference to a user with a few minutes of time in a coffee shop or a waiting room. Even at 10 Mb/s, resume latency is a factor of 3 faster (12 seconds rather than 39 seconds). The user only pays a small price for these substantial gains: he has to carry a portable storage device, and has to wait for the device to be updated at suspend. With a Hi-Speed USB device this wait is just a few seconds.

	No Lookaside	With Lookaside	Win
100 Mb/s	173 (9)	161 (28)	6.9%
10 Mb/s	370 (14)	212 (12)	42.7%
1 Mb/s	2688 (39)	1032 (31)	61.6%
100 Kb/s	30531 (1490)	9530 (141)	68.8%

This table gives the total operation latency (in seconds) for the CDA benchmark at different bandwidths, with and without lookaside to a DVD. Each data point is the mean of three trials, with standard deviation in parentheses. Approximately 50% of the client cache misses were satisfied by lookaside on the DVD. The files on the DVD correspond to the image of a freshly-installed virtual machine, prior to user customization.

Figure 8. Total Operation Latency

To explore the impact of lookaside caching on total operation latency, we constructed a DVD with the VM state captured after installation of Windows XP and the Microsoft Office suite, but before any user-specific or benchmark-specific customizations. We used this DVD as a lookaside device after resume. In a real-life deployment, we expect that an entity such as the computing services organization of a company, university or ISP would create a set of VM installation images and matching DVDs for its clientele. Distributing DVDs to each ISR site does not compromise ease of management because misplaced or missing DVDs do not hurt correctness. A concerned user could, of course, carry his own DVD.

Figure 8 shows that lookaside caching reduces total operation latency at all bandwidths, with the reduction being most noticeable at low bandwidths. Figure 12 shows the distribution of *slowdown* for individual operations in the benchmark. We define slowdown as $(T_{\mathrm{BW}} - T_{\mathrm{NoISR}})/T_{\mathrm{NoISR}}$, with T_{BW} being the benchmark running time at the given bandwidth and T_{NoISR} its running time in VMware without ISR. The figure confirms that lookaside caching reduces the number of operations with very large slowdowns.

6.3 Trace Replay

6.3.1 Benchmark Description

Finally, we used the trace replay benchmark described by Flinn *et al.* [6] in their evaluation of data staging. This benchmark consists of four traces that were obtained from single-user workstations and that range in collection duration from slightly less than 8 hours to slightly more than a day. Figure 9 summarizes the attributes of these traces. To ensure a heavy workload, we replayed these traces as fast as possible, without any filtering or think delays.

6.3.2 Experimental Setup

The experimental setup used was the same as that described in Section 6.1.2.

Trace	Bandwidth	Lookaside Device State			
		No Device	100%	66%	33%
Purcell	100 Mb/s	50.1 (2.6)	53.1 (2.4)	50.5 (3.1)	48.8 (1.9)
	10 Mb/s	61.2 (2.0)	55.0 (6.5)	56.5 (2.9)	56.6 (4.6)
	1 Mb/s	292.8 (4.1)	178.4 (3.1)	223.5 (1.8)	254.2 (2.0)
	100 Kb/s	2828.7 (28.0)	1343.0 (0.7)	2072.1 (30.8)	2404.6 (16.3)
Messiaen	100 Mb/s	26.4 (1.6)	31.8 (0.9)	29.8 (0.9)	27.9 (0.8)
	10 Mb/s	36.3 (0.5)	34.1 (0.7)	36.7 (1.5)	37.8 (0.5)
	1 Mb/s	218.9 (1.2)	117.8 (0.9)	157.0 (0.6)	184.8 (1.3)
	100 Kb/s	2327.3 (14.8)	903.8 (1.4)	1439.8 (6.3)	1856.6 (89.2)
Robin	100 Mb/s	30.0 (1.6)	34.3 (3.1)	33.1 (1.2)	30.6 (2.1)
	10 Mb/s	37.3 (2.6)	33.3 (3.8)	33.8 (2.5)	37.7 (4.5)
	1 Mb/s	229.1 (3.4)	104.1 (1.3)	143.2 (3.3)	186.7 (2.5)
	100 Kb/s	2713.3 (1.5)	750.4 (5.4)	1347.6 (29.6)	2033.4 (124.6)
Berlioz	100 Mb/s	8.2 (0.3)	8.9 (0.2)	9.0 (0.3)	8.8 (0.2)
	10 Mb/s	12.9 (0.8)	9.3 (0.3)	9.9 (0.4)	12.0 (1.6)
	1 Mb/s	94.0 (0.3)	30.2 (0.6)	50.8 (0.3)	71.6 (0.5)
	100 Kb/s	1281.2 (54.6)	216.8 (0.5)	524.4 (0.4)	1090.5 (52.6)

The above results show how long it took for each trace to complete at different portable device states as well as different bandwidth settings. The column labeled "No Device" shows the time taken for trace execution when no portable device was present and all data had to be fetched over the network. The column labeled 100% shows the results when all of the required data was present on the storage device and only meta-data (i.e. `stat` information) was fetched across the network. The rest of the columns show the cases where the lookaside device had varying fractions of the working set. Each data point is the mean of three trials; standard deviations are in parentheses.

Figure 10. Time for Trace Replay

Trace	Number of Operations	Length (Hours)	Update Ops.	Working Set (MB)
purcell	87739	27.66	6%	252
messiaen	44027	21.27	2%	227
robin	37504	15.46	7%	85
berlioz	17917	7.85	8%	57

This table summarizes the file system traces used for the benchmark described in Section 6.3. "Update Ops." only refer to the percentage of operations that change the file system state such as *mkdir*, *close-after-write*, etc. but not individual reads and writes. The working set is the size of the data accessed during trace execution.

Figure 9. Trace Statistics

6.3.3 Results

The performance metric in this benchmark is the time taken for trace replay completion. Although no think time is included, trace replay time is still a good indicator of performance seen by the user.

To evaluate the performance in relation to the portable device state, we varied the amount of data found on the device. This was done by examining the pre-trace snapshots of the traced file systems and then selecting a subset of the trace's working set. For each trace, we began by randomly selecting 33% of the files from the pre-trace snapshot as the initial portable device state. Files were again randomly added to raise the percentage to 66% and then finally 100%. However, these percentages do not necessarily mean that the data from every file present on the portable storage device was used during the benchmark. The snapshot creation tool also creates files that might be overwritten, unlinked, or simply `stat`-ed. Therefore, while these files might be present on the portable device, they would not be read from it during trace replay.

Figure 10 presents our results. The baseline for comparison, shown in column 3 of the figure, was the time taken for trace replay when no lookaside device was present. At the lowest bandwidth (100 Kb/s), the win due to lookaside caching with an up-to-date device was impressive: ranging from 83% for the Berlioz trace (improving from 1281.2 seconds to 216.8 seconds) to 53% for the Purcell trace (improving from 2828.7 seconds to 1343.0 seconds). Even with devices that only had 33% of the data, we were still able to get wins ranging from 25% for the Robin trace to 15% for the Berlioz and Purcell traces.

At a bandwidth of 1 Mb/s, the wins still remain substantial. For an up-to-date device, they range from 68% for the Berlioz trace (improving from 94.0 seconds to 30.2 seconds) to 39% for the Purcell trace (improving from 292.8 seconds to 178.4 seconds). Even when the device contain less useful data, the wins still range from 24% to 46% when the device has 66% of the snapshot and from 13% to 24% when the device has 33% of the snapshot.

On a slow LAN (10 Mb/s) the wins can be strong for an up-to-date device: ranging from 28% for the Berlioz trace (improving from 12.9 seconds to 9.3 seconds) to 6% for Messiaen (improving from 36.3 seconds to 34.1 seconds). Wins tend to tail off beyond this point as the device contains lesser fractions of the working set but it is important to note that performance is never significantly below that of the baseline.

Only on a fast LAN (100 Mb/s) does the overhead of lookaside caching begin to dominate. For an up-to-date device, the traces show a loss ranging from 6% for Purcell (changing from 50.1 seconds to 53.1 seconds) to a loss of 20% for Messiaen (changing from 26.4 seconds to 31.8 seconds). While the percentages might be high, the absolute difference in number of seconds is not and might be imperceptible to the user. It is also interesting to note that the loss decreases when there are fewer files on the portable storage device. For example, the loss for the Robin trace drops from 14% when the device is up-to-date (difference of 4.3 seconds) to 2% when the device has 33% of the files present in the snapshot (difference of 0.6 seconds). As mentioned earlier in Section 6.1.3, the system should suppress lookaside in such scenarios.

Even with 100% success in lookaside caching, the 100 Kb/s numbers for all of the traces are substantially greater than the corresponding 100 Mb/s numbers. This is due to the large number of meta-data accesses, each incurring RPC latency.

7 Broader Uses of Lookaside Caching

Although motivated by portable storage, lookaside caching has the potential to be applied in many other contexts. Any source of data that is hash-addressable can be used for lookaside. Distributed hash tables (DHTs) are one such source. There is growing interest in DHTs such as Pastry [23], Chord [29], Tapestry [33] and CAN [22]. There is also growing interest in planetary-scale services such as PlanetLab [19] and logistical storage such as the Internet Backplane Protocol [2]. Finally, hash-addressable storage hardware is now available [5]. Together, these trends suggest that *Content-Addressable Storage (CAS)* will become a widely-supported service in the future.

Lookaside caching enables a conventional distributed file system based on the client-server model to take advantage of the geographical distribution and replication of data offered by CAS providers. As with portable storage, there is no compromise of the consistency model. Lookaside to a CAS provider improves performance without any negative consequences.

We have recently extended the prototype implementation described in Section 5 to support off-machine CAS providers. Experiments with this extended prototype confirm its performance benefits. For the ISR benchmark described in Section 6.2, Figure 11 shows the performance benefit of using a LAN-attached CAS provider with same contents as the DVD of Figure 8. Since the CAS provider is on a faster machine than the file server, Figure 11 shows a substantial benefit even at 100 Mb/s.

Another potential application of lookaside caching is in implementing a form of *cooperative caching* [1, 4]. A collection of distributed file system clients with mutual trust (typically at one location) can export each other's file caches as CAS providers. No protocol is needed to maintain mutual cache consistency; divergent caches may, at worst, reduce lookaside performance improvement. This form of cooperative caching can be especially valuable in situations where the clients have LAN connectivity to each other, but poor connectivity to a distant file server. The heavy price of a cache miss on a large file is then borne only by the first client to access the file. Misses elsewhere are serviced at LAN speeds, provided the file has not been replaced in the first client's cache.

	No Lookaside	With Lookaside	Win
100 Mb/s	173 (9)	103 (3.9)	40.1%
10 Mb/s	370 (14)	163 (2.9)	55.9%
1 Mb/s	2688 (39)	899 (26.4)	66.6%
100 Kb/s	30531 (1490)	8567 (463.9)	71.9%

This table gives the total operation latency (in seconds) for the CDA benchmark of Section 6.2 at different bandwidths, with and without lookaside to a LAN-attached CAS provider. The CAS provider contains the same state as the DVD used for the results of Figure 8. Each data point is the mean of three trials, with standard deviation in parentheses.

Figure 11. Off-machine Lookaside

8 Conclusion

"Sneakernet," the informal term for manual transport of data, is alive and well today in spite of advances in networking and distributed file systems. Early in this paper, we examined why this is the case. Carrying your data on a portable storage device gives you full confidence that you will be able to access that data anywhere, regardless of network quality, network or server outages, and machine configuration. Unfortunately, this confidence comes at a high price. Remembering to carry the right device, ensuring that data on it is current, tracking updates by collaborators, and guarding against loss, theft and damage are all burdens borne by the user. Most harried mobile users would gladly delegate these chores if only they could be confident that they would have easy access to their critical data at all times and places.

Lookaside caching suggests a way of achieving this goal. Let the true home of your data be in a distributed file system. Make a copy of your critical data on a portable storage device. If you find yourself needing to access the data in a desperate situation, just use the device directly — you are no worse off than if you were relying on sneakernet. In all other situations, use the device for lookaside caching. On a slow network or with a heavily loaded server, you will benefit from improved performance. With network or server outages, you will benefit from improved availability if your distributed file system supports disconnected operation and if you have hoarded all your meta-data.

Notice that you make the decision to use the device di-

rectly or via lookaside caching at the point of use, not *a priori*. This preserves maximum flexibility up front, when there may be uncertainty about the exact future locations where you will need to access the data. Lookaside caching thus integrates portable storage devices and distributed file systems in a manner that combines their strengths. It preserves the intrinsic advantages of performance, availability and ubiquity possessed by portable devices, while simultaneously preserving the consistency, robustness and ease of sharing/collaboration provided by distributed file systems.

One can envision many extensions to lookaside caching. For example, the client cache manager could track portable device state and update stale files automatically. This would require a binding between the name space on the device and the name space of the distributed file system. With this change, a portable device effectively becomes an extension of the client's cache. Another extension would be to support lookaside on individual blocks of a file rather than a whole-file basis. While this is conceptually more general, it is not clear how useful it would be in practice because parts of files would be missing if the portable device were to be used directly rather than via lookaside.

Overall, we believe that the current design of lookaside caching represents a sweet spot in the space of design tradeoffs. It is conceptually simple, easy to implement, and tolerant of human error. It provides good performance and availability benefits without compromising the strengths of portable storage devices or distributed file systems. A user no longer has to choose between distributed and portable storage. You can cache as well as carry!

9 Acknowledgments

We would like to thank Shafeeq Sinnamohideen for implementing the support for off-machine CAS providers and Casey Helfrich for his help with the Internet Suspend/Resume experiments.

References

[1] ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., WANG, R. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles* (Copper Mountain, CO, December 1995).

[2] BECK, M., MOORE, T., PLANK, J.S. An End-to-End Approach to Globally Scalable Network Storage. In *Proceedings of the ACM SIGCOMM Conference* (Pittsburgh, PA, August 2002).

[3] BOLOSKY, W.J., DOUCEUR, J.R., ELY, D., THEIMER, M. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Proceedings of the ACM SIGMETRICS Conference* (Santa Clara, CA, June 2000).

[4] DAHLIN, M.D., WANG, R.Y., ANDERSON, T.E., PATTERSON, D.A. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the USENIX 1994 Operating Systems Design and Implementation Conference* (Monterey, CA, November 1994).

[5] EMC CORP. *EMC Centera Content Addressed Storage System*, 2003. http://www.emc.com/.

[6] FLINN, J., SINNAMOHIDEEN, S., TOLIA, N., AND SATYANARAYANAN, M. Data Staging on Untrusted Surrogates. In *Proceedings of the FAST 2003 Conference on File and Storage Technologies* (March 31 - April 2, 2003).

[7] 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 (February 1988).

[8] HUGHES, J.F., THOMAS, B.W. *Novell's Guide to NetWare 6 Networks*. John Wiley & Sons, 2002.

[9] KOZUCH, M., SATYANARAYANAN, M. Internet Suspend/Resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications* (Calicoon, NY, June 2002).

[10] MANBER, U. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference* (San Fransisco, CA, January 1994), pp. 1–10.

[11] MENEZES, A.J., VAN OORSCHOT, P.C., VANSTONE, S.A. *Handbook of Applied Cryptography*. CRC Press, 2001.

[12] MIGO, FORWARD SOLUTIONS. http://4migo.com/.

[13] MOGUL, J. C., CHAN, Y. M., AND KELLY, T. Design, Implementation, and Evaluation of Duplicate Transfer Detection in HTTP. In *Proceedings of the First Symposium on Networked Systems Design and Implementation* (San Francisco, CA, March 2004).

[14] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Chateau Lake Louise, Banff, Canada, Oct. 2001).

[15] NIST. Secure Hash Standard (SHS). In *FIPS Publication 180-1* (1995).

[16] NIST NET. http://snad.ncsl.nist.gov/itg/nistnet/.

[17] OLSON, M.A., BOSTIC, K., SELTZER, M. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference* (Monterey, CA, June 1999).

[18] PALUSKA, J. M., SAFF, D., YEH, T., AND CHEN, K. Footloose: A Case for Physical Eventual Consistency and Selective Conflict Resolution. In *Proceedings of the Fifth IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, October 2003).

[19] PETERSON, L., ANDERSON, T., CULLER, D., ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the First ACM Workshop on Hot Topics in Networks* (Princeton, NJ, October 2002).

[20] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (January 2002).

[21] RABIN, M. Fingerprinting by Random Polynomials. In *Harvard University Center for Research in Computing Technology Technical Report TR-15-81* (1981).

[22] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., SHENKER, S. A Scalable Content-addressable Network. In *Proceedings of the ACM SIGCOMM Conference* (San Diego, CA, August 2001).

[23] ROWSTRON, A., DRUSCHEL, P. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (Heidelberg, Germany, November 2001).

[24] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., LYON, B. Design and Implementation of the Sun Network File System. In *Summer Usenix Conference Proceedings* (Portland, OR, June 1985).

[25] SOBTI, S., GARG, N., ZHENG, F., LAI, J., SHAO, Y., ZHANG, C., ZISKIND, E., KRISHNAMURTHY, A., AND WANG, R. Segank: A Distributed Mobile Storage System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies* (San Francisco, CA, March 31 - April 2, 2004).

[26] SOBTI, S., GARG, N., ZHANG, C., YU, X., KRISHNAMURTHY, A., WANG, R. PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In *Proceedings of the First USENIX Conference on File and Storage Technologies* (Monterey, CA, Jan 2002).

[27] SPRING, N. T., AND WETHERALL, D. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of ACM SIGCOMM* (August 2000).

[28] STEINER, J.G., NEUMAN, C., SCHILLER, J.I. Kerberos: An Authentication Service for Open Network Systems. In *USENIX Conference Proceedings* (Dallas, TX, Winter 1988).

[29] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M.F., BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001* (San Diego, CA, August 2001).

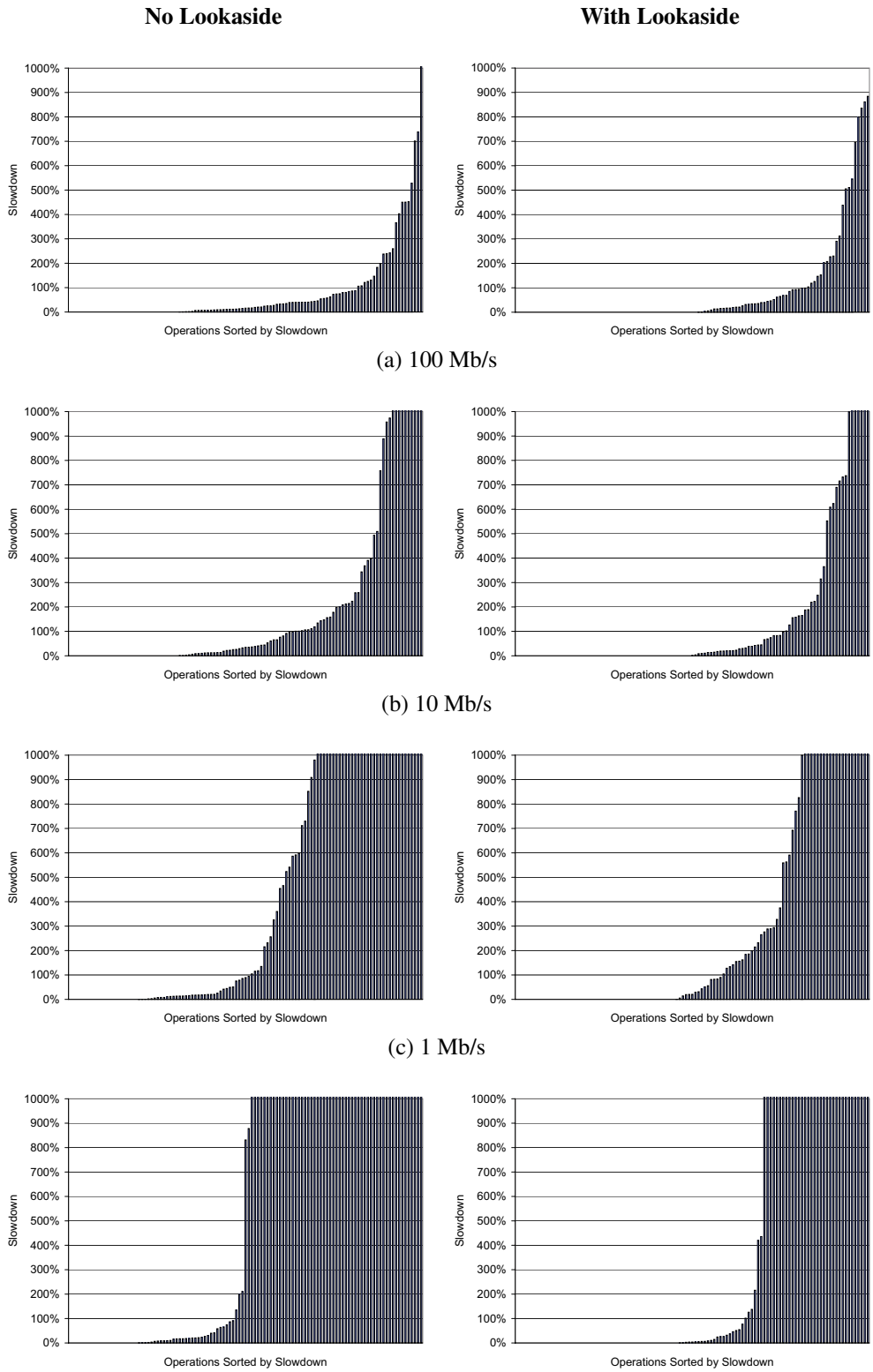
[30] TERRY, D.B. Caching Hints in Distributed Systems. *IEEE Transactions on Software Engineering 13*, 1 (January 1987).

[31] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., BRESSOUD, T., PERRIG, A. Opportunistic Use of Content-Addressable Storage for Distributed File Systems. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, TX, June 2003).

[32] WANT, R., PERING, T., DANNEELS, G., KUMAR, M., SUNDAR, M., AND LIGHT, J. The Personal Server: Changing the Way We Think About Ubiquitous Computing. In *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing* (Goteborg, Sweden, 2002).

[33] ZHAO, B.Y., KUBATOWICZ, J., JOSEPH, A. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Tech. Rep. UCB/CSD-01-1141, University of California at Berkeley, April 2001.

[34] ZWICKY, E.D., COOPER, S., CHAPMAN, D.B. *Building Internet Firewalls, Second Edition*. O'Reilly & Associates, Inc., 2000, ch. 17.4: File Sharing for Microsoft Networks.



These figures compares the distribution of slowdown for the operations of the CDA benchmark without lookaside caching to their slowdowns with lookaside caching to a DVD.

Figure 12. Impact of Lookaside Caching on Slowdown of CDA Benchmark Operations