A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications

TYLER HARTER, CHRIS DRAGGA, MICHAEL VAUGHN,
ANDREA C. ARPACI-DUSSEAU, and REMZI H. ARPACI-DUSSEAU,
University of Wisconsin-Madison

We analyze the I/O behavior of *iBench*, a new collection of productivity and multimedia application workloads. Our analysis reveals a number of differences between iBench and typical file-system workload studies, including the complex organization of modern files, the lack of pure sequential access, the influence of underlying frameworks on I/O patterns, the widespread use of file synchronization and atomic operations, and the prevalence of threads. Our results have strong ramifications for the design of next generation local and cloud-based storage systems.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms: Measurement, Performance

Additional Key Words and Phrases: Application performance, file systems, measurement, trace study

ACM Reference Format:
Harter, T., Dragga, C., Vaughn, M., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2012. A file is not a file: Understanding the I/O behavior of Apple desktop applications. ACM Trans. Comput. Syst. 30, 3, Article 10 (August 2012), 39 pages.
DOI = 10.1145/2324876.2324878 http://doi.acm.org/10.1145/2324876.2324878

1. INTRODUCTION

The design and implementation of file and storage systems has long been at the forefront of computer systems research. Innovations such as namespace-based locality [McKusick et al. 1984], crash consistency via journaling [Hagmann 1987; Prabhakaran et al. 2005a] and copy-on-write [Bonwick and Moore 2007; Rosenblum and Ousterhout 1992], checksums and redundancy for reliability [Bartlett and Spainhower 2004; Bonwick and Moore 2007; Patterson et al. 1988; Prabhakaran et al. 2005b], scalable on-disk structures [Sweeney et al. 1996], distributed file systems [Howard et al. 1988; Sandberg 1985], and scalable cluster-based storage systems [Decandia et al. 2007; Ghemawat et al. 2003; Lee and Thekkath 1996] have greatly influenced how data is managed and stored within modern computer systems.

This material was based on work supported by the National Science Foundation under CSR-1017518 as well as by generous donations from Network Appliance and Google. T. Harter and C. Dragga were supported by the Guri Sohi Fellowship and the David DeWitt Fellowship, respectively. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.
Authors' address: T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, Department of Computer Sciences, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706-1685; email: tylerharter@gmail.com.

DOI 10.1145/2324876.2324878 http://doi.acm.org/10.1145/2324876.2324878

Much of this work in file systems over the past three decades has been shaped by *measurement*: the deep and detailed analysis of workloads [Baker et al. 1991; Douceur and Bolosky 1999; Ellard and Seltzer 2003; Howard et al. 1988; Leung et al. 2008; Ousterhout et al. 1985; Roselli et al. 2000; Satyanarayanan 1981; Vogels 1999]. One excellent example is found in work on the Andrew File System [Howard et al. 1988]; detailed analysis of an early AFS prototype led to the next-generation protocol, including the key innovation of callbacks. Measurement helps us understand the systems of today so we can build improved systems for tomorrow.

Whereas most studies of file systems focus on the corporate or academic intranet, most file-system users work in the more mundane environment of the home, accessing data via desktop PCs, laptops, and compact devices such as tablet computers and mobile phones. Despite the large number of previous studies, little is known about home-user applications and their I/O patterns.

Home-user applications are important today, and their importance will increase as more users store data not only on local devices but also in the cloud. Users expect to run similar applications across desktops, laptops, and phones; therefore, the behavior of these applications will affect virtually every system with which a user interacts. I/O behavior is especially important to understand since it greatly impacts how users perceive overall system latency and application performance [Endo et al. 1994].

While a study of how users typically exercise these applications would be interesting, the first step is to perform a detailed study of I/O behavior under typical but controlled workload tasks. This style of *application study*, common in the field of computer architecture [Woo et al. 1995], is different from the *workload study* found in systems research, and can yield deeper insight into how the applications are constructed and how file and storage systems need to be designed in response.

Home-user applications are fundamentally large and complex, containing millions of lines of code [Macintosh Business Unit (Microsoft) 2006]. In contrast, traditional UNIX-based applications are designed to be simple, to perform one task well, and to be strung together to perform more complex tasks [Ritchie and Thompson 1973]. This modular approach of UNIX applications has not prevailed [Lampson 1999]: modern applications are standalone monoliths, providing a rich and continuously evolving set of features to demanding users. Thus, it is beneficial to study each application individually to ascertain its behavior.

In this article, we present the first in-depth analysis of the I/O behavior of modern home-user applications; we focus on productivity applications (for word processing, spreadsheet manipulation, and presentation creation) and multimedia software (for digital music, movie editing, and photo management). Our analysis centers on two Apple software suites: iWork, consisting of Pages, Numbers, and Keynote; and iLife, which contains iPhoto, iTunes, and iMovie. As Apple's market share grows [Tilmann 2010], these applications form the core of an increasingly popular set of workloads; as device convergence continues, similar forms of these applications are likely to access user files from both stationary machines and moving cellular devices. We call our collection the *iBench task suite*.

To investigate the I/O behavior of the iBench suite, we build an instrumentation framework on top of the powerful DTrace tracing system found inside Mac OS X [Cantrill et al. 2004]. DTrace allows us not only to monitor system calls made by each traced application, but also to examine stack traces, in-kernel functions such as page-ins and page-outs, and other details required to ensure accuracy and completeness. We also develop an application harness based on AppleScript [Apple Computer, Inc. 2011] to drive each application in the repeatable and automated fashion that is key to any study of GUI-based applications [Endo et al. 1994].

Our careful study of the tasks in the iBench suite has enabled us to make a number of interesting observations about how applications access and manipulate stored data. In addition to confirming standard past findings (e.g., most files are small; most bytes accessed are from large files [Baker et al. 1991]), we find the following new results.

A File Is Not a File. Modern applications manage large databases of information organized into complex directory trees. Even simple word-processing documents, which appear to users as a "file", are in actuality small file systems containing many sub-files (e.g., a Microsoft .doc file is actually a FAT file system containing pieces of the document). File systems should be cognizant of such hidden structure in order to lay out and access data in these complex files more effectively.

Sequential Access Is Not Sequential. Building on the trend noticed by Vogels [1999] for Windows NT, we observe that even for streaming media workloads, "pure" sequential access is increasingly rare. Since file formats often include metadata in headers, applications often read and re-read the first portion of a file before streaming through its contents. Prefetching and other optimizations might benefit from a deeper knowledge of these file formats.

Auxiliary Files Dominate. Applications help users create, modify, and organize content, but user files represent a small fraction of the files touched by modern applications. Most files are helper files that applications use to provide a rich graphical experience, support multiple languages, and record history and other metadata. File-system placement strategies might reduce seeks by grouping the hundreds of helper files used by an individual application.

Writes Are Often Forced. As the importance of home data increases (e.g., family photos), applications are less willing to simply write data and hope it is eventually flushed to disk. We find that most written data is explicitly forced to disk by the application; for example, iPhoto calls `fsync` thousands of times in even the simplest of tasks. For file systems and storage, the days of delayed writes [Mogul 1994] may be over; new ideas are needed to support applications that desire durability.

Renaming Is Popular. Home-user applications commonly use atomic operations, in particular `rename`, to present a consistent view of files to users. For file systems, this may mean that transactional capabilities [Olson 2007] are needed. It may also necessitate a rethinking of traditional means of file locality; for example, placing a file on disk based on its parent directory [McKusick et al. 1984] does not work as expected when the file is first created in a temporary location and then renamed.

Multiple Threads Perform I/O. Virtually all of the applications we study issue I/O requests from a number of threads; a few applications launch I/Os from hundreds of threads. Part of this usage stems from the GUI-based nature of these applications; it is well known that threads are required to perform long-latency operations in the background to keep the GUI responsive [Ousterhout 1995]. Thus, file and storage systems should be thread-aware so they can better allocate bandwidth.

Frameworks Influence I/O. Modern applications are often developed in sophisticated IDEs and leverage powerful libraries, such as Cocoa and Carbon. Whereas UNIX-style applications often directly invoke system calls to read and write files, modern libraries put more code between applications and the underlying file system; for example, including `"cocoa.h"` in a Mac application imports 112,047 lines of code from 689 different files [Pike 2010]. Thus, the behavior of the framework, and not just the

application, determines I/O patterns. We find that the default behavior of some Cocoa APIs induces extra I/O and possibly unnecessary (and costly) synchronizations to disk. In addition, use of different libraries for similar tasks within an application can lead to inconsistent behavior between those tasks. Future storage design should take these libraries and frameworks into account.

This article contains four major contributions. First, we describe a general tracing framework for creating benchmarks based on interactive tasks that home users may perform (e.g., importing songs, exporting video clips, saving documents). Second, we deconstruct the I/O behavior of the tasks in iBench; we quantify the I/O behavior of each task in numerous ways, including how directories are used, the types of files accessed (e.g., counts and sizes), the access patterns (e.g., read/write, sequentiality, and preallocation), caching considerations (e.g., data reuse and prefetching hints), transactional properties (e.g., durability, atomicity, and isolation), and threading. Third, we describe how these qualitative changes in I/O behavior may impact the design of future systems. Finally, we present the 34 traces from the iBench task suite; by making these traces publicly available and easy to use, we hope to improve the design, implementation, and evaluation of the next generation of local and cloud storage systems:

`http://www.cs.wisc.edu/adsl/Traces/ibench`

This article is an expansion of our earlier work [Harter et al. 2011]. We have made two major additions to our quantitative analysis: a section on directory I/O (Section 4.1), and a section on memory-related behavior (Section 4.4). We have also added minor sections which discuss memory-mapped I/O (Section 4.3.1), read and write request sizes (Section 4.3.3), open durations (Section 4.3.6), metadata access (Section 4.3.7), and file locking (Section 4.5.3). The sequential-access section (Section 4.3.4) has also been expanded to provide details about full-file access.

The remainder of this article is organized as follows. We begin by presenting a detailed timeline of the I/O operations performed by one task in the iBench suite; this motivates the need for a systematic study of home-user applications. We next describe our methodology for creating the iBench task suite. We then spend the majority of the paper quantitatively analyzing the I/O characteristics of the full iBench suite. Finally, we summarize the implications of our findings on file-system design.

2. CASE STUDY

The I/O characteristics of modern home-user applications are distinct from those of UNIX applications studied in the past. To motivate the need for a new study, we investigate the complex I/O behavior of a single representative task. Specifically, we report in detail the I/O performed over time by the Pages (4.0.3) application, a word processor, running on Mac OS X Snow Leopard (10.6.2) as it creates a blank document, inserts 15 JPEG images each of size 2.5 MB, and saves the document as a Microsoft .doc file.

Figure 1 shows the I/O this task performs (see the caption for a description of the symbols used). The top portion of the figure illustrates the accesses performed over the full lifetime of the task: at a high level, it shows that more than 385 files spanning six different categories are accessed by eleven different threads, with many intervening calls to `fsync` and `rename`. The bottom portion of the figure magnifies a short time interval, showing the reads and writes performed by a single thread accessing the primary .doc productivity file. From this one experiment, we illustrate each finding described in the introduction. We first focus on the single access that saves the user's document (bottom), and then consider the broader context surrounding this file save, where we observe a flurry of accesses to hundreds of helper files (top).

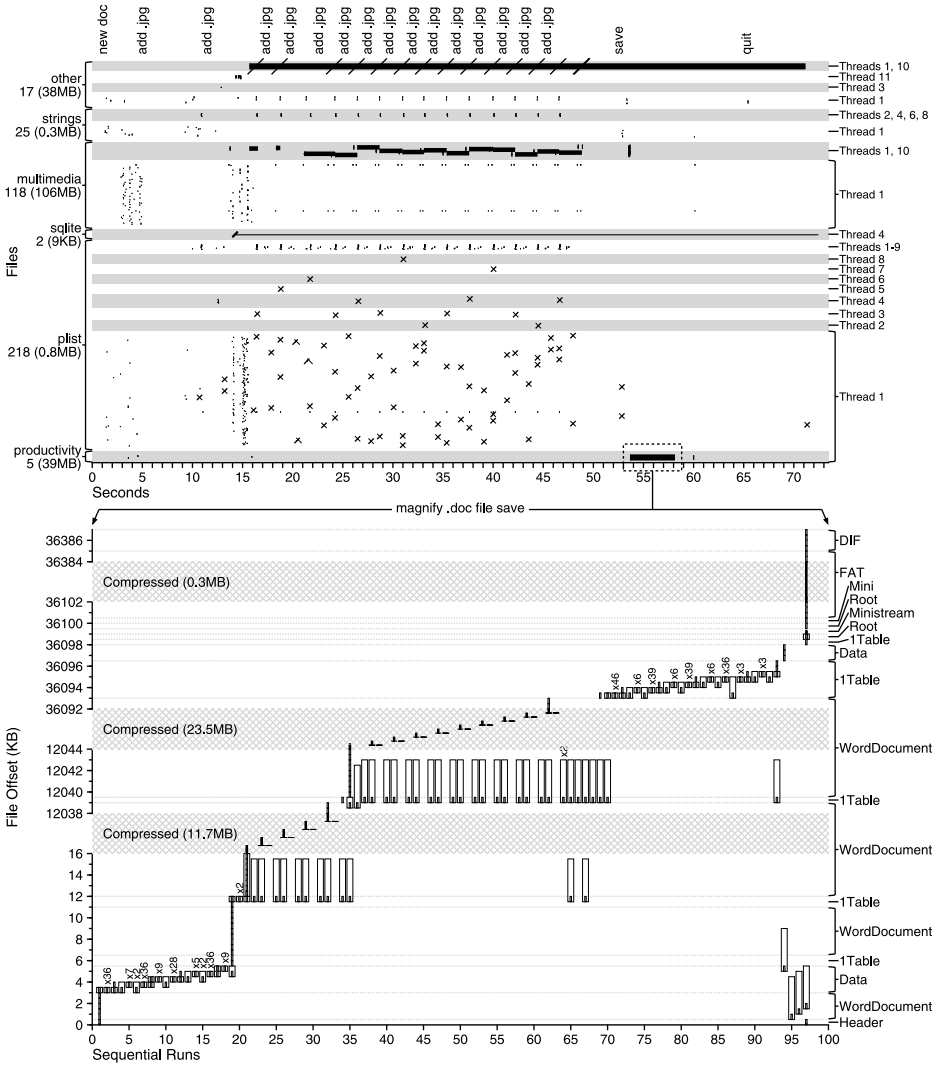

Fig. 1. Pages saving a word document. The top graph shows the 75-second timeline of the entire run, while the bottom graph is a magnified view of seconds 54 to 58. In the top graph, annotations on the left categorize files by type and indicate file count and amount of I/O; annotations on the right show threads. Black bars are file accesses (reads and writes), with thickness logarithmically proportional to bytes of I/O. / is an `fsync`; \ is a `rename`; X is both. In the bottom graph, individual reads and writes to the .doc file are shown. Vertical bar position and bar length represent the offset within the file and number of bytes touched. Thick white bars are reads; thin gray bars are writes. Repeated runs are marked with the number of repetitions. Annotations on the right indicate the name of each file section.

A File Is Not a File. Focusing on the magnified timeline of reads and writes to the productivity .doc file, we see that the file format comprises more than just a simple file. Microsoft .doc files are based on the FAT file system and allow bundling of multiple files in the single .doc file. This .doc file contains a directory (Root), three streams for large data (WordDocument, Data, and 1Table), and a stream for small data (Ministream). Space is allocated in the file with three sections: a file allocation table (FAT), a double-indirect FAT (DIF) region, and a ministream allocation region (Mini).

Sequential Access Is Not Sequential. The complex FAT-based file format causes random access patterns in several ways: first, the header is updated at the beginning and end of the magnified access; second, data from individual streams is fragmented throughout the file; and third, the 1Table stream is updated before and after each image is appended to the WordDocument stream.

Auxiliary Files Dominate. Although saving the single .doc we have been considering is the sole purpose of this task, we now turn our attention to the top timeline and see that 385 different files are accessed. There are several reasons for this multitude of files. First, Pages provides a rich graphical experience involving many images and other forms of multimedia; together with the 15 inserted JPEGs, this requires 118 multimedia files. Second, users want to use Pages in their native language, so application text is not hard-coded into the executable but is instead stored in 25 different .strings files. Third, to save user preferences and other metadata, Pages uses a SQLite database (2 files) and a number of key-value stores (218 .plist files).

Writes Are Often Forced; Renaming Is Popular. Pages uses both of these actions to enforce basic transactional guarantees. It uses `fsync` to flush write data to disk, making it durable; it uses `rename` to atomically replace old files with new files so that a file never contains inconsistent data. The timeline shows these invocations numerous times. First, Pages regularly uses `fsync` and `rename` when updating the key-value store of a .plist file. Second, `fsync` is used on the SQLite database. Third, for each of the 15 image insertions, Pages calls `fsync` on a file named "tempData" (classified as "other") to update its automatic backup.

Multiple Threads Perform I/O. Pages is a multi-threaded application and issues I/O requests from many different threads during the experiment. Using multiple threads for I/O allows Pages to avoid blocking while I/O requests are outstanding. Examining the I/O behavior across threads, we see that Thread 1 performs the most significant portion of I/O, but ten other threads are also involved. In most cases, a single thread exclusively accesses a file, but it is not uncommon for multiple threads to share a file.

Frameworks Influence I/O. Pages was developed in a rich programming environment where frameworks such as Cocoa or Carbon are used for I/O; these libraries impact I/O patterns in ways the developer might not expect. For example, although the application developers did not bother to use `fsync` or `rename` when saving the user's work in the .doc file, the Cocoa library regularly uses these calls to atomically and durably update relatively unimportant metadata, such as "recently opened" lists stored in .plist files. As another example, when Pages tries to read data in 512-byte chunks from the .doc, each read goes through the `STDIO` library, which only reads in 4 KB chunks. Thus, when Pages attempts to read one chunk from the 1Table stream, seven unrequested chunks from the WordDocument stream are also incidentally read (offset 12039 KB). In other cases, regions of the .doc file are repeatedly accessed unnecessarily. For example, around the 3 KB offset, read/write pairs occur dozens of times. Pages uses a library to write 2-byte words; each time a word is written, the library reads, updates, and writes back an entire 512-byte chunk. Finally, we see evidence of redundancy between libraries: even though Pages has a backing SQLite database for some of its properties, it also uses .plist files, which function across Apple applications as generic property stores.

This one detailed experiment has shed light on a number of interesting I/O behaviors that indicate that home-user applications are indeed different than traditional workloads. A new workload suite is needed that accurately reflects these applications.

3. IBENCH TASK SUITE

Our goal in constructing the iBench task suite is two-fold. First, we would like iBench to be representative of the tasks performed by home users. For this reason, iBench contains popular applications from the iLife and iWork suites for entertainment and productivity. Second, we would like iBench to be relatively simple for others to use for file and storage system analysis. For this reason, we automate the interactions of a home user and collect the resulting traces of I/O system calls. The traces are available online at this site: `http://www.cs.wisc.edu/adsl/Traces/ibench`. We now describe in more detail how we met these two goals.

3.1. Representative

To capture the I/O behavior of home users, iBench models the actions of a "reasonable" user interacting with iPhoto, iTunes, iMovie, Pages, Numbers, and Keynote. Since the research community does not yet have data on the exact distribution of tasks that home users perform, iBench contains tasks that we believe are common and uses files with sizes that can be justified for a reasonable user. iBench contains 34 different tasks, each representing a home user performing one distinct operation. If desired, these tasks could be combined to create more complex workflows and I/O workloads. The six applications and corresponding tasks are as follows.

iLife iPhoto 8.1.1 (419). Digital photo album and photo manipulation software. iPhoto stores photos in a library that contains the data for the photos (which can be in a variety of formats, including JPG, TIFF, and PNG), a directory of modified files, a directory of scaled down images, and two files of thumbnail images. The library stores metadata in a SQLite database. iBench contains six tasks exercising user actions typical for iPhoto: starting the application and importing, duplicating, editing, viewing, and deleting photos in the library. These tasks modify both the image files and the underlying database. Each of the iPhoto tasks operates on 400 2.5-MB photos, representing a user who has imported 12 megapixel photos (2.5 MB each) from a full 1-GB flash card on his or her camera.

iLife iTunes 9.0.3 (15). A media player capable of both audio and video playback. iTunes organizes its files in a private library and supports most common music formats (e.g., MP3, AIFF, WAVE, AAC, and MPEG-4). iTunes does not employ a database, keeping media metadata and playlists in both a binary and an XML file. iBench contains five tasks for iTunes: starting iTunes, importing and playing an album of MP3 songs, and importing and playing an MPEG-4 movie. Importing requires copying files into the library directory and, for music, analyzing each song file for gapless playback. The music tasks operate over an album (or playlist) of ten songs while the movie tasks use a single 3-minute movie.

iLife iMovie 8.0.5 (820). Video editing software. iMovie stores its data in a library that contains directories for raw footage and projects, and files containing video footage thumbnails. iMovie supports both MPEG-4 and Quicktime files. iBench contains four tasks for iMovie: starting iMovie, importing an MPEG-4 movie, adding a clip from this movie into a project, and exporting a project to MPEG-4. The tasks all use a 3-minute movie because this is a typical length for home videos on video-sharing websites.

iWork Pages 4.0.3 (766). A word processor. Pages uses a ZIP-based file format and can export to DOC, PDF, RTF, and basic text. iBench includes eight tasks for Pages: starting up, creating and saving, opening, and exporting documents with and without images and with different formats. The tasks use 15-page documents.

Table I. 34 Tasks of the iBench Suite

		Name	Description	Files	(MB)	Acc.	(MB)	RD%	WR%	Acc./s	MB/s
iLife	iPhoto	Start	Open iPhoto with library of 400 photos	779	(336.7)	828	(25.4)	78.8	21.2	**151.1**	4.6
		Imp	Import 400 photos into empty library	5900	(1966.9)	8709	(3940.3)	74.4	25.6	26.7	**12.1**
		Dup	Duplicate 400 photos from library	2928	(1963.9)	5736	(2076.2)	52.4	47.6	**237.9**	**86.1**
		Edit	Sequentially edit 400 photos from library	12119	(4646.7)	18927	(12182.9)	69.8	30.2	19.6	**12.6**
		Del	Sequentially del. 400 photos; empty trash	15246	(23.0)	15247	(25.0)	21.8	78.2	**280.9**	0.5
		View	Sequentially view 400 photos	2929	(1006.4)	3347	(1005.0)	98.1	1.9	24.1	**7.2**
	iTunes	Start	Open iTunes with 10 song album	143	(184.4)	195	(9.3)	54.7	45.3	**72.4**	3.4
		ImpS	Import 10 song album to library	68	(204.9)	139	(264.5)	66.3	33.7	**75.2**	**143.1**
		ImpM	Import 3 minute movie to library	41	(67.4)	57	(42.9)	48.0	52.0	**152.4**	**114.6**
		PlayS	Play album of 10 songs	61	(103.6)	80	(90.9)	96.9	3.1	0.4	0.5
		PlayM	Play 3 minute movie	56	(77.9)	69	(32.0)	92.3	7.7	2.2	1.0
	iMovie	Start	Open iMovie with 3 minute. clip in project	433	(223.3)	786	(29.4)	99.9	0.1	**134.8**	**5.0**
		Imp	Import 3 minute .m4v (20MB) to ''Events''	184	(440.1)	383	(122.3)	55.6	44.4	29.3	**9.3**
		Add	Paste 3 min. clip from ''Events'' to project	210	(58.3)	547	(2.2)	47.8	52.2	**357.8**	1.4
		Exp	Export 3 minute video clip	70	(157.9)	546	(229.9)	55.1	44.9	2.3	1.0
iWork	Pages	Start	Open Pages	218	(183.7)	228	(2.3)	99.9	0.1	**97.7**	1.0
		New	Create 15 text-page doc; save as .pages	135	(1.6)	157	(1.0)	73.3	26.7	**50.8**	0.3
		NewP	Create 15 JPG doc; save as .pages	408	(112.0)	997	(180.9)	60.7	39.3	**54.6**	**9.9**
		Open	Open 15 text page document	103	(0.8)	109	(0.6)	99.5	0.5	**57.6**	0.3
		PDF	Export 15 page document as .pdf	107	(1.5)	115	(0.9)	91.0	9.0	**41.3**	0.3
		PDFP	Export 15 JPG document as .pdf	404	(77.4)	965	(110.9)	67.4	32.6	**49.7**	**5.7**
		DOC	Export 15 page document as .doc	112	(1.0)	121	(1.0)	87.9	12.1	**44.4**	0.4
		DOCP	Export 15 JPG document as .doc	385	(111.3)	952	(183.8)	61.1	38.9	**46.3**	**8.9**
	Numbers	Start	Open Numbers	283	(179.9)	360	(2.6)	99.6	0.4	**115.5**	0.8
		New	Save 5 sheets/col graphs as .numbers	269	(4.9)	313	(2.8)	90.7	9.3	9.6	0.1
		Open	Open 5 sheet spreadsheet	119	(1.3)	137	(1.3)	99.8	0.2	**48.7**	0.5
		XLS	Export 5 sheets/column graphs as .xls	236	(4.6)	272	(2.7)	94.9	5.1	8.5	0.1
	Keynote	Start	Open Keynote	517	(183.0)	681	(1.1)	99.8	0.2	**229.8**	0.4
		New	Create 20 text slides; save as .key	637	(12.1)	863	(5.4)	92.4	7.6	**129.1**	0.8
		NewP	Create 20 JPG slides; save as .key	654	(92.9)	901	(103.3)	66.8	33.2	**70.8**	**8.1**
		Play	Open/play presentation of 20 text slides	318	(11.5)	385	(4.9)	99.8	0.2	**95.0**	1.2
		PlayP	Open/play presentation of 20 JPG slides	321	(45.4)	388	(55.7)	69.6	30.4	**72.4**	**10.4**
		PPT	Export 20 text slides as .ppt	685	(12.8)	918	(10.1)	78.8	21.2	**115.2**	1.3
		PPTP	Export 20 JPG slides as .ppt	723	(110.6)	996	(124.6)	57.6	42.4	**61.0**	**7.6**

The table summarizes the 34 tasks of iBench, specifying the application, a short name for the task, and a longer description of the actions modeled. The I/O is characterized according to the number of files read or written, the sum of the maximum sizes of all accessed files, the number of file accesses that read or write data (Acc.), the number of bytes read or written, the percentage of I/O bytes that are part of a read (or write; RD% and WR%, respectively), and the rate of I/O per CPU-second in terms of both file accesses and bytes (Acc./s and MB/s, respectively). Each core is counted individually, so at most 2 CPU-seconds can be counted per second on our dual-core test machine. CPU utilization is measured with the UNIX `top` utility, which in rare cases produces anomalous CPU utilization snapshots; those values are ignored.

iWork Numbers 2.0.3 (332). A spreadsheet application. Numbers organizes its files with a ZIP-based format and exports to XLS and PDF. The four iBench tasks for Numbers include starting Numbers, generating a spreadsheet and saving it, opening the spreadsheet, and exporting a spreadsheet to XLS. To model a possible user working on a budget, tasks utilize a five-page spreadsheet with one column graph per sheet.

iWork Keynote 5.0.3 (791). A presentation and slideshow application. Keynote saves to a .key ZIP-based format and exports to Microsoft's PPT format. The seven iBench tasks for Keynote include starting Keynote, creating slides with and without images, opening and playing presentations, and exporting to PPT. Each Keynote task uses a 20-slide presentation.

Table I contains a brief description and basic I/O characteristics for each of our 34 iBench tasks. The table illustrates that the iBench tasks perform a significant amount of I/O. Most tasks access hundreds of files, which in aggregate contain tens or hundreds of megabytes of data. The tasks perform widely differing amounts of I/O, from less than a megabyte to more than a gigabyte. Most tasks perform many more

reads than writes. Finally, the tasks exhibit high I/O throughput, often transferring tens of megabytes of data per CPU-second.

3.2. Easy to Use

To enable other system evaluators to easily use these tasks, the iBench suite is packaged as a set of 34 system call traces. To ensure reproducible results, the 34 user tasks were first automated with AppleScript, a general-purpose GUI scripting language. AppleScript provides generic commands to emulate mouse clicks through menus and application-specific commands to capture higher-level operations. Application-specific commands bypass a small amount of I/O by skipping dialog boxes; however, we use them whenever possible for expediency.

The system call traces were gathered using DTrace [Cantrill et al. 2004], a kernel and user level dynamic instrumentation tool. DTrace is used to instrument the entry and exit points of all system calls dealing with the file system; it also records the current state of the system and the parameters passed to and returned from each call.

While tracing with DTrace was generally straightforward, we addressed four challenges in collecting the iBench traces. First, file sizes are not always available to DTrace; thus, we record every file's initial size and compute subsequent file size changes caused by system calls such as `write` or `ftruncate`. Second, iTunes uses the `ptrace` system call to disable tracing; we circumvent this block by using `gdb` to insert a breakpoint that automatically returns without calling `ptrace`. Third, the `volfs` pseudo-file system in HFS+ (Hierarchical File System) allows files to be opened via their inode number instead of a file name; to include pathnames in the trace, we instrument the `build_path` function to obtain the full path when the task is run. Fourth, tracing system calls misses I/O resulting from memory-mapped files; therefore, we purged memory and instrumented kernel page-in functions to measure the amount of memory-mapped file activity. We found that the amount of memory-mapped I/O is negligible in most tasks; we thus do not include this I/O in the iBench traces or analysis, except for a brief discussion in Section 4.3.1.

To provide reproducible results, the traces must be run on a single file-system image. Therefore, the iBench suite also contains snapshots of the initial directories to be restored before each run; initial state is critical in file-system benchmarking [Agrawal et al. 2009].

4. ANALYSIS OF IBENCH TASKS

The iBench task suite enables us to study the I/O behavior of a large set of home-user actions. As shown from the timeline of I/O behavior for one particular task in Section 2, these tasks are likely to access files in complex ways. To characterize this complex behavior in a quantitative manner across the entire suite of 34 tasks, we focus on answering six categories of questions.

— How often are directories accessed? How much data is read from them?
— What different types of files are accessed and what are the sizes of these files?
— How are files accessed? Does I/O occur via system calls or via memory-mapped files? How are files accessed for reads and writes? How many bytes are accessed per request? Are files accessed sequentially? Is space preallocated?
— Is the data repeatedly accessed? Are patterns conducive to caching/buffering data? Are there caching/prefetching hints?
— What are the transactional properties? Are writes flushed with `fsync` or performed atomically? Are file locks used for isolation?
— How do multithreaded applications distribute I/O across different threads?

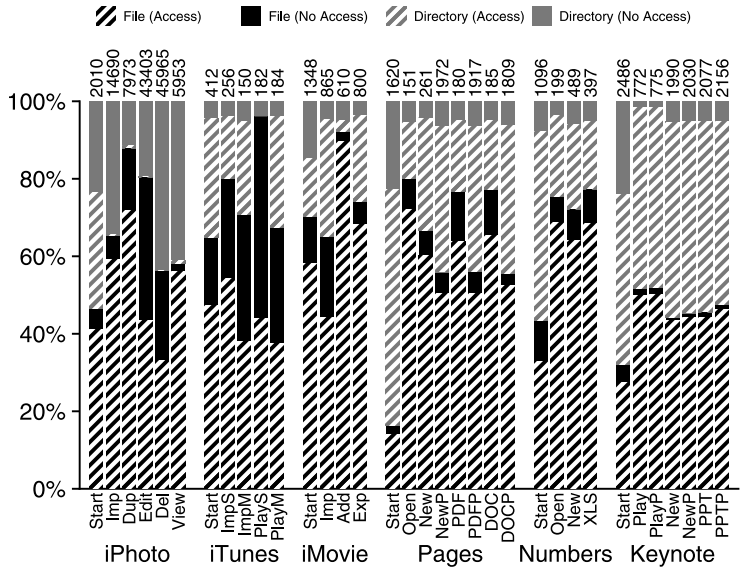

Fig. 2. Types of opens. This plot divides opens into four types—file opens that access the file's data, file opens that do not, directory opens that read the directory, and directory opens that do not—and displays the relative size of each. The numbers atop the bars indicate the total number of opens in each task.

Answering these questions has two benefits. First, the answers can guide file and storage system developers to target their systems better to home-user applications. Second, the characterization will help users of iBench to select the most appropriate traces for evaluation and to understand their resulting behavior.

All measurements were performed on a Mac Mini running Mac OS X Snow Leopard version 10.6.2 and the HFS+ file system. The machine has 2 GB of memory and a 2.26 GHz Intel Core Duo processor.

4.1. Open Types: Directories and Files

One of the first measurements necessary for workload analysis is the ratio of file operations to directory operations. The results inform where to concentrate investigations into the rest of the workload; if data access is skewed towards either files or directories, it would be best to focus on that.

We categorize opens into four groups based on their target and their usage: opens of files that access the files' contents, opens of files that do not access the contents of the files directly, opens of directories that read their contents, and opens of directories that do not read them. We display our results in Figure 2.

Our results show that opens to files are generally more common than directory opens, though significant variance exists between tasks. In particular, iMovie Add uses over 90% of its opens for file I/O, while Pages Start uses less than 20% of them for this purpose. Opens to files that access data outnumber those that do not across all tasks; only a few of the iPhoto and iTunes tasks open more than 10–15% of their files without accessing them. Though some opens that do not access files result in no further manipulation of their target file, investigating our traces shows that many of these opens are used for locking, calls to `fsync` or `mmap`, and metadata access.

While directory opens usually occur less frequently than file opens, they nonetheless have a significant presence, particularly for iWork. Their quantity varies widely, ranging from over 60% for Keynote Start to under 5% for iTunes PlayS. As with file opens,

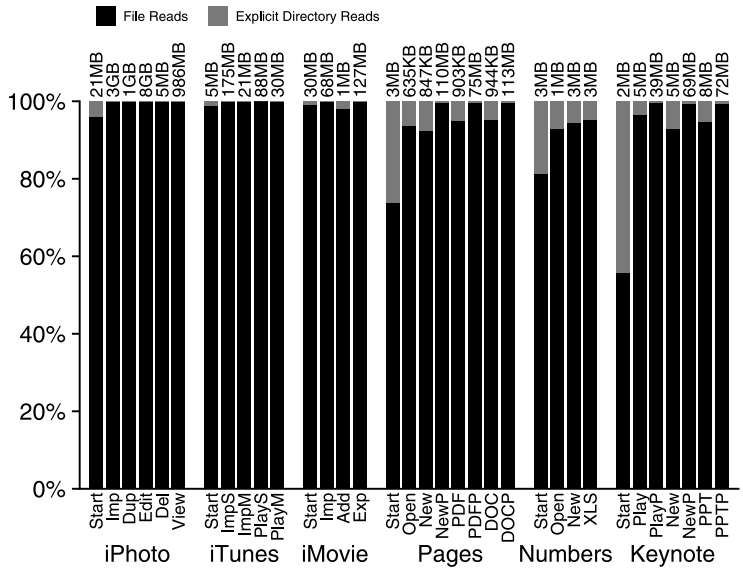

Fig. 3. Distribution of bytes between file and directory reads. This plot shows how bytes read are distributed between files and explicit reads of directories. The numbers atop the bars indicate the total number of bytes read.

the majority of directory opens access directory entries, although all iPhoto workloads other than Start access very little data from the directories they open. Similarly, directory opens that do not explicitly access directory entries are not necessarily useless or used solely to confirm the existence of a given directory; many examine metadata attributes of their children or of the directory itself or change the working directory.

Given that a sizable minority of opens are on directories, we next examine how many bytes are read from each type of object in Figure 3. This plot omits the implicit directory reads that occur during pathname resolution when files are opened.

Despite the prevalence of directory opens, directory reads account for almost none of the I/O performed in the iLife suite and usually comprise at most 5–7% of the total I/O in iLife. The only exceptions are the Start tasks for Keynote, Pages, and Numbers, where directory reads account for roughly 50%, 25%, and 20%, respectively, of all data read. None of these tasks read much data, however, and, in general, the proportion of data read from directories diminishes rapidly as the total data read for the task rises.

Summary. While the iBench applications frequently open directories, the vast majority of reads are from files; the total data read from directories in any given task never exceeds 1 MB. Thus, we focus on file I/O for the rest of our analysis.

4.2. Nature of Files

We now characterize the high-level behavior of the iBench tasks. In particular, we study the different types and sizes of files opened by each iBench task.

4.2.1. File Types. The iLife and iWork applications store data across a variety of files in a number of different formats; for example, iLife applications tend to store their data in libraries (or data directories) unique to each user, while iWork applications

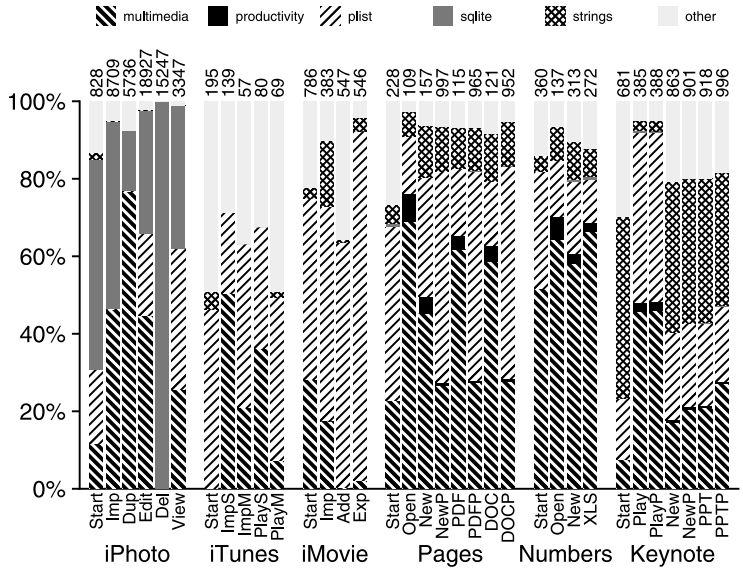

Fig. 4. Types of files accessed by number of opens. This plot shows the relative frequency with which file descriptors are opened upon different file types. The number at the end of each bar indicates the total number of unique file descriptors opened on files.

organize their documents in proprietary ZIP-based files. The extent to which tasks access different types of files greatly influences their I/O behavior.

To understand accesses to different file types, we place each file into one of six categories, based on file name extensions and usage. *Multimedia* files contain images (e.g., JPEG), songs (e.g., MP3, AIFF), and movies (e.g., MPEG-4). *Productivity* files are documents (e.g., .pages, DOC, PDF), spreadsheets (e.g., .numbers, XLS), and presentations (e.g., .key, PPT). *SQLite* files are database files. *Plist* files are property-list files in XML containing key-value pairs for user preferences and application properties. *Strings* files contain strings for localization of application text. Finally, *Other* contains miscellaneous files such as plain text, logs, files without extensions, and binary files.

Figure 4 shows the frequencies with which tasks open and access files of each type; most tasks perform hundreds of these accesses. Multimedia file opens are common in all workloads, though they seldom predominate, even in the multimedia-heavy iLife applications. Conversely, opens of productivity files are rare, even in iWork applications that use them; this is likely because most iWork tasks create or view a single productivity file. Because .plist files act as generic helper files, they are relatively common. SQLite files only have a noticeable presence in iPhoto, where they account for a substantial portion of the observed opens. Strings files occupy a significant minority of most workloads (except iPhoto and iTunes). Finally, between 5% and 20% of files are of type "Other" (except for iTunes, where they are more prevalent).

Figure 5 displays the percentage of I/O bytes accessed for each file type. In bytes, multimedia I/O dominates most of the iLife tasks, while productivity I/O has a significant presence in the iWork tasks; file descriptors on multimedia and productivity files tend to receive large amounts of I/O. SQLite, Plist, and Strings files have a smaller share of the total I/O in bytes relative to the number of opened files; this implies that tasks access only a small quantity of data for each of these files opened (e.g., several key-value pairs in a .plist). In most tasks, files classified as "Other" receive a more significant portion of the I/O (the exception is iTunes).

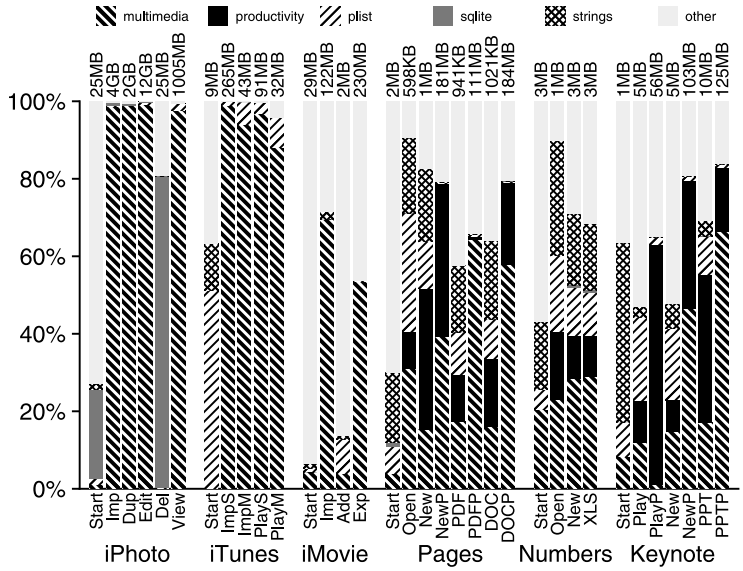

Fig. 5. Types of files opened by I/O size. This plot shows the relative frequency with which each task performs I/O upon different file types. The number atop each bar indicates the total bytes of I/O accessed.

Summary. Home applications access a wide variety of file types, generally opening multimedia files the most frequently. iLife tasks tend to access bytes primarily from multimedia or files classified as "Other"; iWork tasks access bytes from a broader range of file types, with some emphasis on productivity files.

4.2.2. File Sizes. Large and small files present distinct challenges to the file system. For large files, finding contiguous space can be difficult, while for small files, minimizing initial seek time is more important. We investigate two different questions regarding file size. First, what is the distribution of file sizes accessed by each task? Second, what portion of accessed bytes resides in files of various sizes?

To answer these questions, we record file sizes when each unique file descriptor is closed. We categorize sizes as very small (<4 KB), small (<64 KB), medium (<1 MB), large (<10 MB), or very large (≥10 MB). We track how many accesses are to files in each category and how many of the bytes belong to files in each category.

Figure 6 shows the number of accesses to files of each size. Accesses to very small files are extremely common, especially for iWork, accounting for over half of all the accesses in every iWork task. Small file accesses have a significant presence in the iLife tasks. The large quantity of very small and small files is due to frequent use of .plist files that store preferences, settings, and other application data; these files often fill just one or two 4 KB pages.

Figure 7 shows the proportion of the files in which the bytes of accessed files reside. Large and very large files dominate every startup workload and nearly every task that processes multimedia files. Small files account for few bytes and very small files are essentially negligible.

Summary. Agreeing with many previous studies (e.g., Baker et al. [1991]), we find that while applications tend to open many very small files (<4 KB), most of the bytes accessed are in large files (>1 MB).

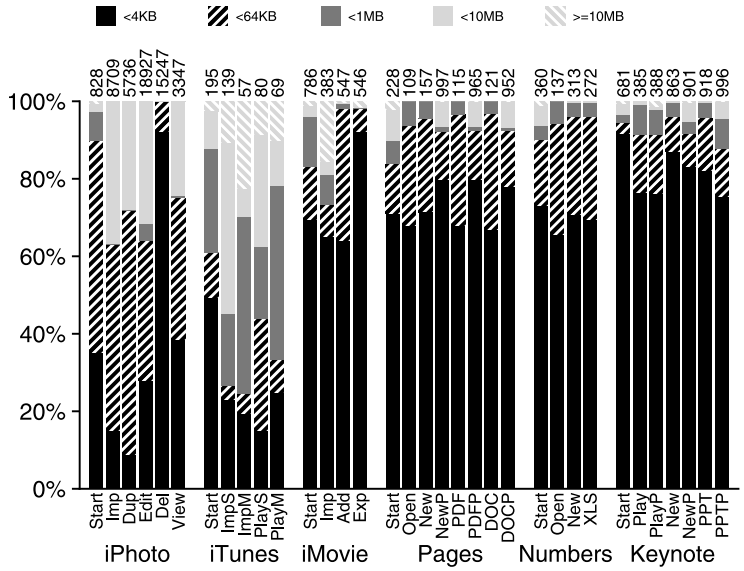

Fig. 6. File sizes, weighted by number of accesses. This graph shows the number of accessed files in each file size range upon access ends. The total number of file accesses appears at the end of the bars. Note that repeatedly-accessed files are counted multiple times, and entire file sizes are counted even upon partial file accesses.

4.3. Access Patterns

We next examine how the nature of file accesses has changed, studying the read and write patterns of home applications. These patterns include whether data is transferred via memory-mapped I/O or through read and write requests; whether files are used for reading, writing, or both; whether files are accessed sequentially or randomly; and finally, whether or not blocks are preallocated via hints to the file system.

4.3.1. I/O Mechanisms. UNIX provides two mechanisms for reading and writing to a file that has been opened. First, calls to `read`, `write`, or similar functions may be performed on a file descriptor (we call this request-based I/O). Second, a process may use `mmap` to map a file into a region of virtual memory and then just access that region. We explore how often these two mechanisms are used.

For each of our tasks, we clear the system's disk cache using the `purge` command so that we can observe all pageins that result from memory-mapped I/O. We are interested in application behavior, not the I/O that results from loading the application, so we filter pageins to executable memory or from files under /System/Library and /Library. We also exclude pageins from a 13 MB file named "/usr/share/icu/icudt40l.dat"; all the applications use this file, so it should typically be resident in memory. Figure 8 shows how frequently pageins occur relative to request-based I/O. We do not observe any pageouts. In general, memory-mapped I/O is negligible. For the two exceptions, Numbers Start and Keynote Start, the pagein traffic is from JPG files.

In the Numbers and Keynote Start tasks, the user is presented with a list of templates from which to choose. A JPG thumbnail represents each template. The applications map these thumbnails into virtual memory.

Summary. The vast majority of I/O is performed by reading and writing to open file descriptors. Only a few of the iBench tasks have significant pageins from

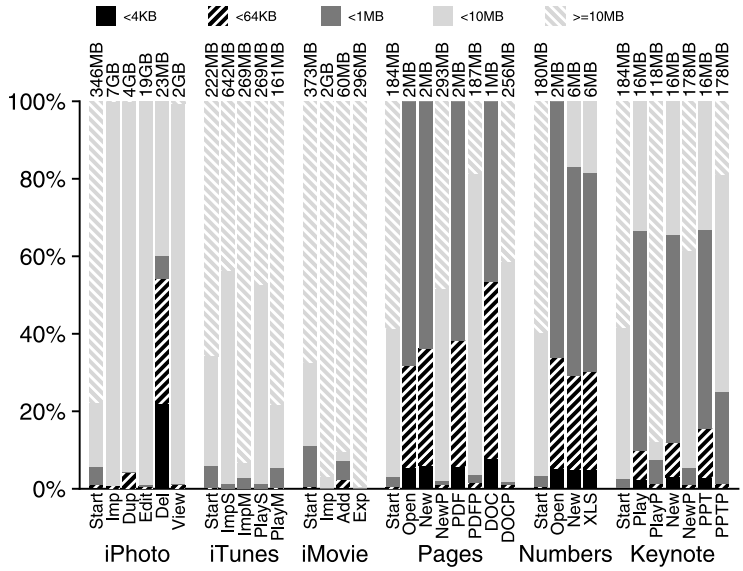

Fig. 7. File sizes, weighted by the bytes in accessed files. This graph shows the portion of bytes in accessed files of each size range upon access ends. The sum of the file sizes appears at the end of the bars. This number differs from total file footprint since files change size over time and repeatedly accessed files are counted multiple times.

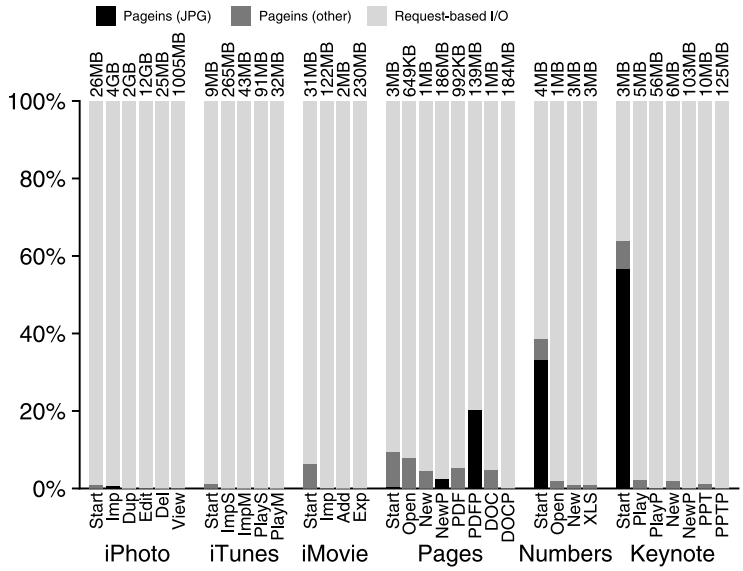

Fig. 8. File access mechanisms. Memory-mapped I/O is compared to request-based I/O. The values atop the bars indicate the sum of both I/O types. Memory-mapped I/O to JPG files is indicated by the black bars.

memory-mapped files; most of this pagein traffic is from images. For the rest of our analysis, we exclude memory-mapped I/O since it is generally negligible.

4.3.2. File Accesses. One basic characteristic of our workloads is the division between reading and writing on open file descriptors. If an application uses an open file only for

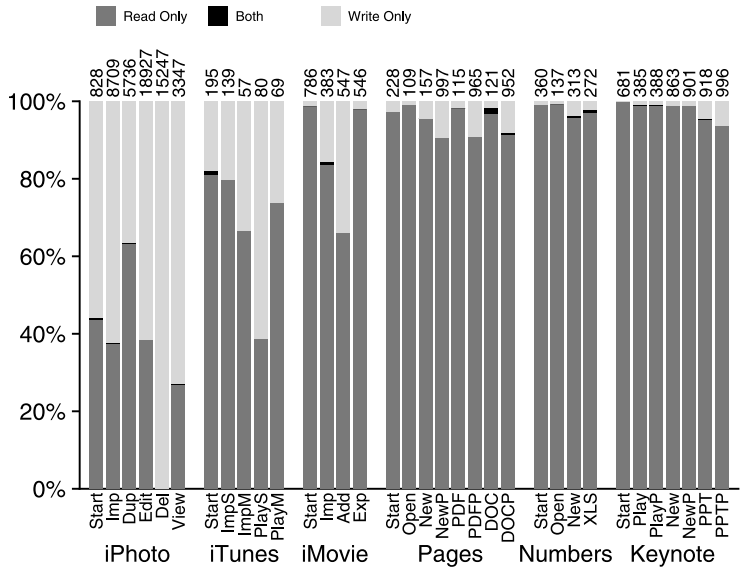

Fig. 9. Read/write distribution by file descriptor. File descriptors can be used only for reads, only for writes, or for both operations. This plot shows the percentage of file descriptors in each category. This is based on usage, not open flags. Any duplicate file descriptors (e.g., created by dup) are treated as one and file descriptors on which the program does not perform any subsequent read or write are ignored.

reading (or only for writing) or performs more activity on file descriptors of a certain type, then the file system may be able to allocate memory and disk space in a more intelligent fashion.

To determine these characteristics, we classify each opened file descriptor based on the types of accesses–read, write, or both read and write–performed during its lifetime. We also ignore the actual flags used when opening the file since we found they do not accurately reflect behavior; in all workloads, almost all write-only file descriptors were opened with `O_RDWR`. We measure both the proportional usage of each type of file descriptor and the relative amount of I/O performed on each.

Figure 9 shows how many file descriptors are used for each type of access. The overwhelming majority of file descriptors are used exclusively for reading or writing; read-write file descriptors are quite uncommon. Overall, read-only file descriptors are the most common across a majority of workloads; write-only file descriptors are popular in some iLife tasks, but are rarely used in iWork.

We observe different patterns when analyzing the amount of I/O performed on each type of file descriptor, as shown in Figure 10. First, although iWork tasks have very few write-only file descriptors, significant write I/O is often performed on those descriptors. Second, even though read-write file descriptors are rare, when present, they account for relatively large portions of total I/O (especially for exports to .doc, .xls, and .ppt).

Summary. While many files are opened with the `O_RDWR` flag, most of them are subsequently accessed write-only; thus, file open flags cannot be used to predict how a file will be accessed. However, when an open file is both read and written by a task, the amount of traffic to that file occupies a significant portion of the total I/O. Finally, the rarity of read-write file descriptors may derive in part from the tendency of applications to write to a temporary file which they then rename as the target file, instead of overwriting the target file; we explore this tendency more in Section 4.5.2.

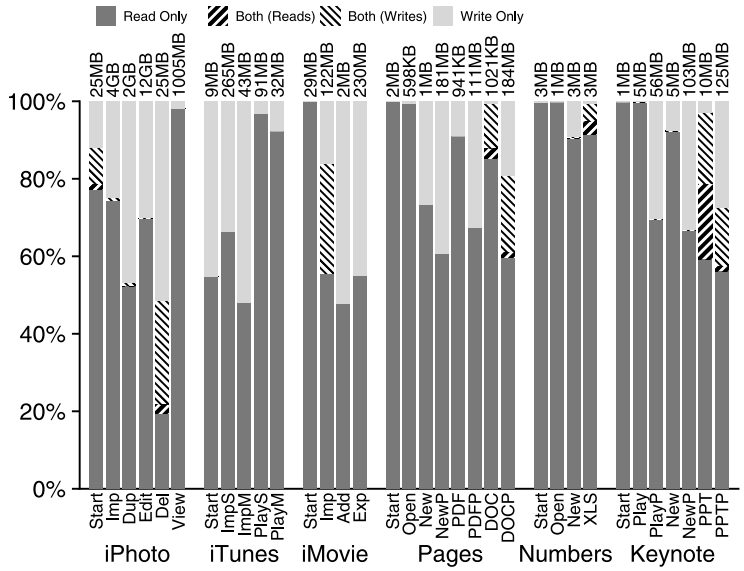

Fig. 10. Read/write distribution by bytes. The graph shows how I/O bytes are distributed among the three access categories. The unshaded dark gray indicates bytes read as a part of read-only accesses. Similarly, unshaded light gray indicates bytes written in write-only accesses. The shaded regions represent bytes touched in read-write accesses, and are divided between bytes read and bytes written.

4.3.3. Read and Write Sizes. Another metric that affects file-system performance is the number and size of individual read and write system calls. If applications perform many small reads and writes, prefetching and caching may be a more effective strategy than it would be if applications tend to read or write entire files in one or two system calls. In addition, these data can augment some of our other findings, especially those regarding file sizes and complex access patterns.

We examine this by recording the number of bytes accessed by each read and write system call. As in our file size analysis in Section 4.2.2, we categorize these sizes into five groups: very small ($\leq$4 KB), small ($\leq$32 KB), medium ($\leq$128 KB), large ($\leq$1 MB), and very large ($>$1 MB).

Figures 11 and 12 display the number of reads and writes, respectively, of each size. We see very small operations dominating the iWork suite, with a more diverse variety of sizes in iLife; in particular, very large writes dominate iTunes ImpS, indicating that the act of copying songs occupies the majority of the I/O operations for that task. Conversely, many iPhoto write tasks are composed almost entirely of very small operations, and reads for iLife tasks are usually dominated by a mixture of small and very small operations.

Figures 13 and 14 show the proportion of accessed bytes for each group. As with file sizes, large and very large reads cover substantial proportions of bytes in most of the workloads across both iLife and iWork. However, while large and very large writes account for sizable proportions of the workloads in iLife and the iWork workloads that use images, the vast majority of bytes in many of the other iWork tasks result from very small writes. Much of this is likely due to the complex patterns we observe when applications write to the complex files frequently used in productivity applications, although some of it also derives from the small amount of bytes that these tasks write.

Summary. We find that applications perform large numbers of very small ($\leq$4 KB) reads and writes, which agrees with our findings in Section 4.2.2 that applications

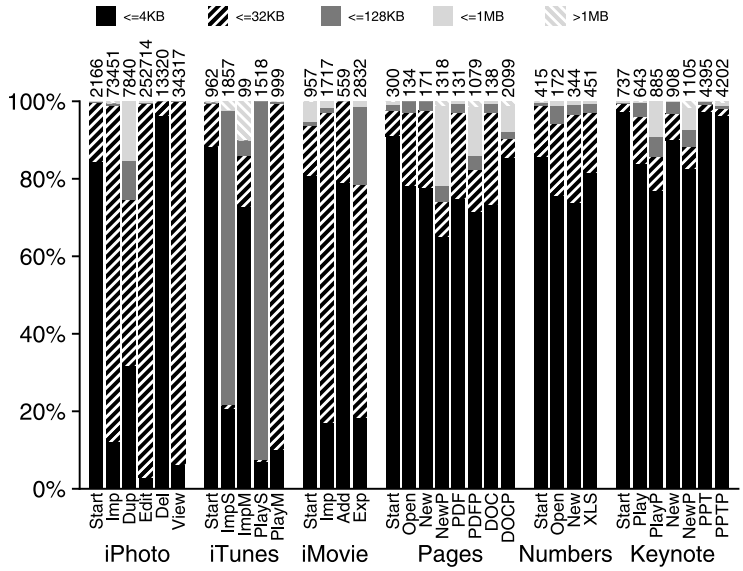

Fig. 11. Read size distribution. This plot groups individual read operations by the number of bytes each operation returns. The numbers atop the bars indicate the total number of read system calls in each task.

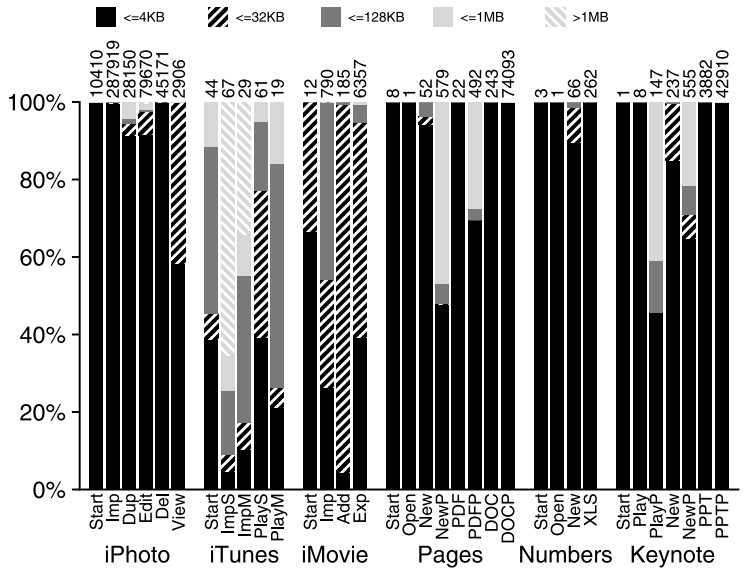

Fig. 12. Write size distribution. This plot groups individual write operations by the number of bytes each operation returns. The numbers atop the bars indicate the number of write system calls in each task.

open many small files. While we also find that applications read most of their bytes in large operations, very small writes dominate many of the iWork workloads, resulting from a combination of the complex access patterns we have observed and the limited write I/O they perform.

4.3.4. Sequentiality. Historically, files have usually been read or written entirely sequentially [Baker et al. 1991]. We next determine whether sequential accesses are

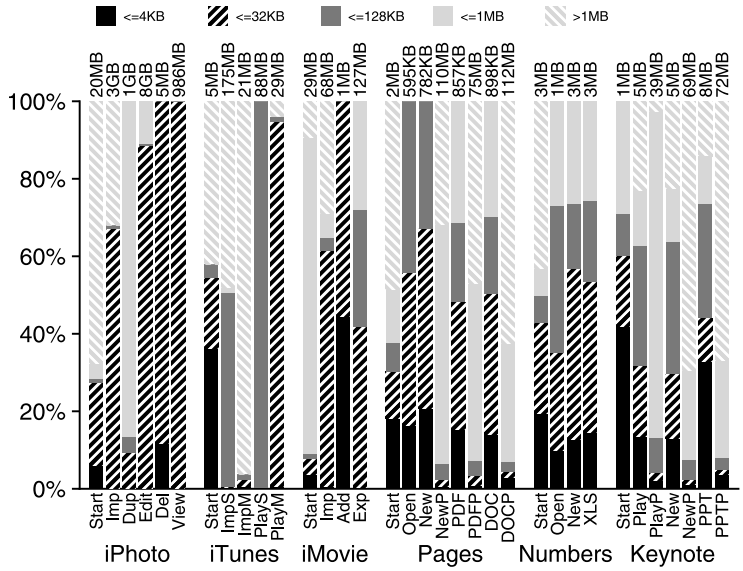

Fig. 13. Read size distribution by bytes. This plot groups individual read operations by the percent of total bytes read for which each call is responsible. The numbers atop the bars indicate the total number of bytes read.

dominant in iBench. We measure this by examining all reads and writes performed on each file descriptor and noting the percentage of files accessed in strict sequential order (weighted by bytes).

We display our measurements for read and write sequentiality in Figures 15 and 16, respectively. The portions of the bars in black indicate the percent of file accesses that exhibit pure sequentiality. We observe high read sequentiality in iWork, but little in iLife (with the exception of the Start tasks and iTunes Import). The inverse is true for writes: while a majority of iLife writes are sequential, iWork writes are seldom sequential outside of Start tasks.

Investigating the access patterns to multimedia files more closely, we note that many iLife applications first touch a small header before accessing the entire file sequentially. To better reflect this behavior, we define an access to a file as "nearly sequential" when at least 95% of the bytes read or written to a file form a sequential run. We found that a large number of accesses fall into the "nearly sequential" category given a 95% threshold; the results do not change much with lower thresholds.

The slashed portions of the bars in Figures 15 and 16 show observed sequentiality with a 95% threshold. Tasks with heavy use of multimedia files exhibit greater sequentiality with the 95% threshold for both reading and writing. In several workloads (mainly iPhoto and iTunes), the I/O classified almost entirely as non-sequential with a 100% threshold is classified as nearly sequential. The difference for iWork applications is much less striking, indicating that accesses are more random.

In addition to this analysis of sequential and random accesses, we also measure how often a completely sequential access reads or writes an entire file. Figure 17 divides sequential reads into those that read the full file and those that only read part of it. In nearly all cases, the access reads the entire file; the only tasks for which sequential accesses of part of the file account for more than five percent of the total are iTunes Start and iMovie Imp and Exp. We omit the corresponding plot for writes, since virtually all sequential writes cover the entire file.

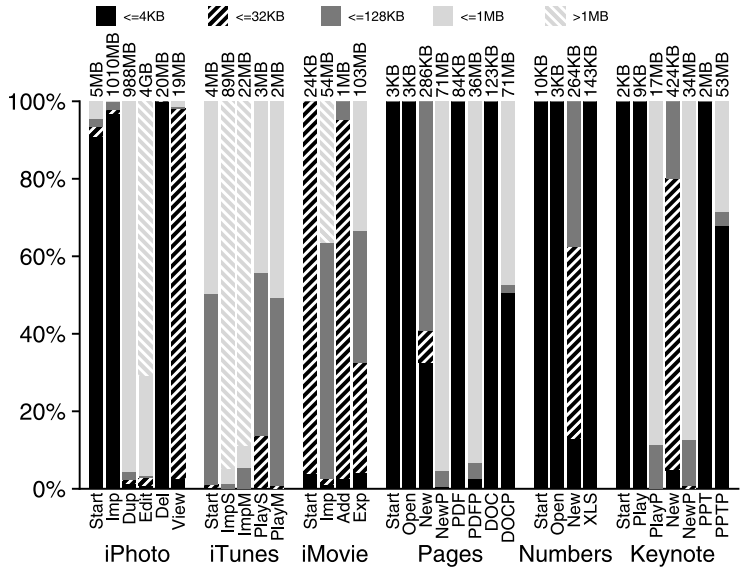

Fig. 14. Write size distribution. This plot groups individual write operations by the percent of total bytes written each for which each call is responsible. The numbers atop the bars indicate the total number of bytes written.

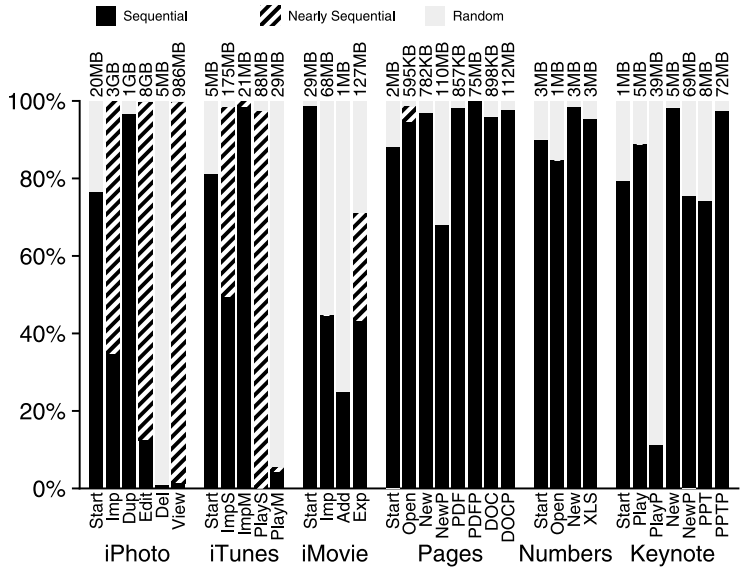

Fig. 15. Read sequentiality. This plot shows the portion of file read accesses (weighted by bytes) that are sequentially accessed.

Summary. A substantial number of tasks contain purely sequential accesses. When the definition of a sequential access is loosened such that only 95% of bytes must be consecutive, then even more tasks contain primarily sequential accesses. These "nearly sequential" accesses result from metadata stored at the beginning of complex multimedia files: tasks frequently touch bytes near the beginning of multimedia files before sequentially reading or writing the bulk of the file. Those accesses that are fully

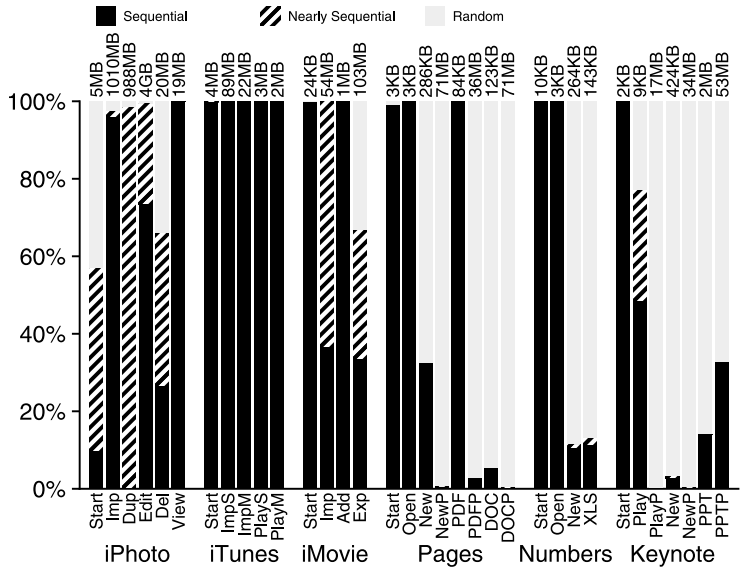

Fig. 16. Write sequentiality. This plot shows the portion of file write accesses (weighted by bytes) that are sequentially accessed.

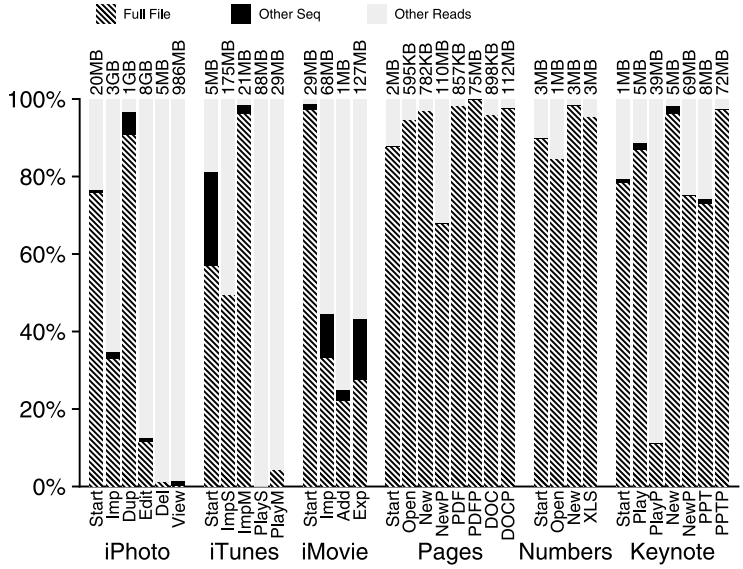

Fig. 17. Full-file sequential reads. This plot divides those reads that are fully sequential into partial reads of the file and reads of the entire file. The number atop each bar shows the total bytes the task read.

sequential tend to access the entire file at once; applications that perform a substantial number of sequential reads of parts of files, like iMovie, often deal with relatively large files that would be impractical to read in full.

4.3.5. Preallocation. One of the difficulties file systems face when allocating contiguous space for files is not knowing how much data will be written to those files. Applications can communicate this information by providing hints [Patterson et al. 1995]

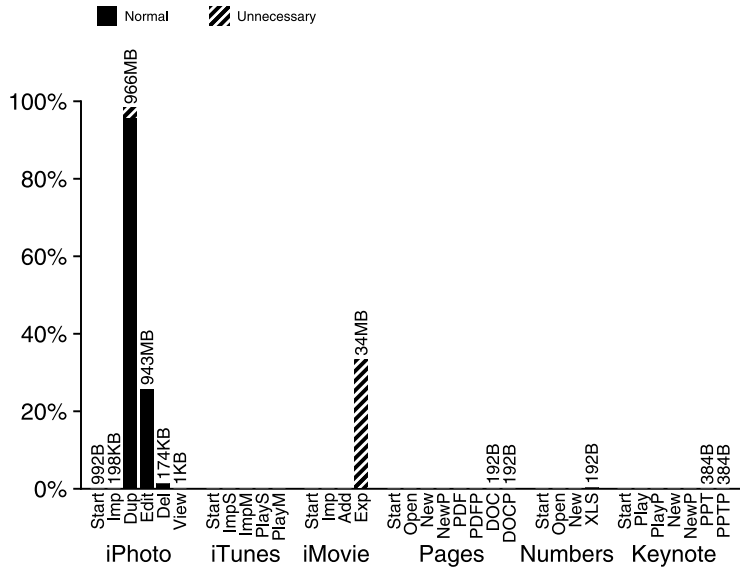

Fig. 18. Preallocation hints. The sizes of the bars indicate which portion of file extensions are preallocations; unnecessary preallocations are diagonally striped. The number atop each bar indicates the absolute amount preallocated.

to the file system to preallocate an appropriate amount of space. In this section, we quantify how often applications use preallocation hints and how often these hints are useful.

We instrument two calls usable for preallocation: `pwrite` and `ftruncate`. `pwrite` writes a single byte at an offset beyond the end of the file to indicate the future end of the file; `ftruncate` directly sets the file size. Sometimes a preallocation does not communicate anything useful to the file system because it is immediately followed by a single write call with all the data; we flag these preallocations as unnecessary.

Figure 18 shows the portion of file growth that is the result of preallocation. In all cases, preallocation was due to calls to `pwrite`; we never observed `ftruncate` preallocation. Overall, applications rarely preallocate space and the preallocations that occur are often useless.

The three tasks with significant preallocation are iPhoto Dup, iPhoto Edit, and iMovie Exp. iPhoto Dup and Edit both call a `copyPath` function in the Cocoa library that preallocates a large amount of space and then copies data by reading and writing it in 1 MB chunks. iPhoto Dup sometimes uses `copyPath` to copy scaled-down images of size 50–100 KB; since these smaller files are copied with a single write, the preallocation does not communicate anything useful. iMovie Exp calls a Quicktime append function that preallocates space before writing the actual data; however, the data is appended in small 128 KB increments. Thus, the append is not split into multiple `write` calls; the preallocation is useless.

Summary. Although preallocation has the potential to be useful, few tasks use it to provide hints, and a significant number of the hints that are provided are useless. The hints are provided inconsistently: although iPhoto and iMovie both use preallocation for some tasks, neither application uses preallocation during import.

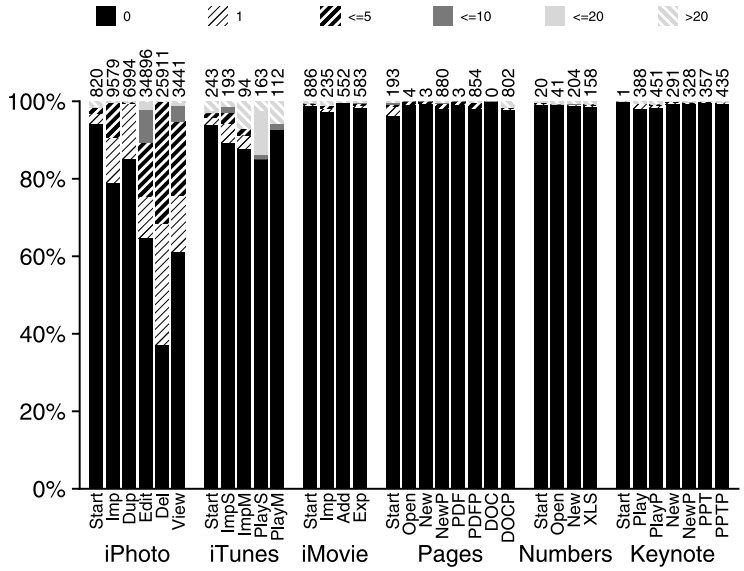

Fig. 19. Lifetime of each file open by intervening opens. This plot groups the file opens present in each task based on the number of opens to other files that occur during their lifetime. The number at the end of each bar displays the largest number of opens during the lifetime of any one open in each task.

4.3.6. Open Durations. The average length of time that an application keeps its files open can be useful to know, as it provides a sense of how many resources the operating system should allocate to that file and its file descriptors. If applications keep file descriptors open for extended periods of time, closing a file descriptor likely means that the application is done with the file and that it no longer needs its contents; whereas if applications quickly close file descriptors after individual operations, the operation may not provide a meaningful hint for caching or buffering. Previous studies have found that files tend to be open for only brief periods of time [Baker et al. 1991], and we wish to determine whether this remains the case.

Due to the automated and compressed nature of our tasks, we cannot obtain a precise sense of how long file descriptors remain open under normal operation. We approximate this, however, by examining the number of operations performed during the lifetime of each file descriptor. Specifically, we assume that most file descriptors remain open for only a short period and examine the number of files opened during the lifetime of each access. As each task usually involves a large number of file descriptors, this should provide a reasonable sense of the relative lifetime of each open. We only track this statistic for file descriptors that access data, though the duration metric counts all intervening file opens.

We display our results in Figure 19. The vast majority of file descriptors in most tasks see no intervening file opens during their lifetimes. The only applications with tasks in which more than 5% of their file descriptors have others opened during their lifetimes are iPhoto and iTunes. Of these, iPhoto has significantly more file descriptors of this type, but most of them see fewer than five opens and many only one. iTunes, on the other hand, has fewer file descriptors with multiple intervening opens, but those that do generally have significantly more intervening opens than those observed in iPhoto. This is particularly apparent in iTunes PlayS, where about 10% of the file descriptors see between 11 and 20 opens during their lifetimes.

Despite the predominance of file descriptors without intervening opens, all experiments except Pages Open, New, PDF, and DOC and Keynote Start feature at least one file open over the lifetime of a large number of other file opens. The nature of this file varies: in iPhoto and iWork, it is generally a database, while in iTunes it is a resource fork of an iTunes-specific file, and in iMovie, it is generally the movie being manipulated (except for Start, where it is a database).

The two applications that open the most files while others are open, iPhoto and iTunes, also make the most use of multi-threaded I/O, as shown in Section 4.6. This indicates that these applications may perform I/O in parallel. Similarly, the prevalence of file descriptors without intervening opens indicates that most I/O operations are likely performed serially.

Summary. In the vast majority of cases, applications close the file they are working on before opening another, though they frequently keep one or two files open for the majority of the task. Those applications that do open multiple file descriptors in succession tend to perform substantial multi-threaded I/O, indicating that the opens may represent parallel access to files instead of longer-lived accesses.

4.3.7. Metadata Access. Reading and writing file data hardly comprises the entirety of file system operations; many file system calls are devoted to manipulating file metadata. As previous studies have noted [Jacob et al. 2000], `stat` calls (including variations like `fstat`, `stat64`, and `lstat`) often occupy large portions of file system workloads, as they provide an easy way to verify whether a file currently exists and to examine the attributes of a file before opening it. Similarly, `statfs` offers an interface to access metadata about the file system on which a given file resides. Mac OS X provides an additional function, `getattrlist`, which combines the data returned in these two calls, as well as providing additional information; some of the attributes that it deals with can be manipulated by the application with the `setattrlist` call. Finally, the `xattr` related calls allow applications to retrieve and set arbitrary key-value metadata on each file. To determine how heavily applications use these metadata operations, we compare the relative frequency of these calls to the frequency of file data accesses.

We display our results in Figure 20. Metadata accesses occur substantially more often than data accesses (i.e., open file descriptors that receive at least one read or write request) with calls to `stat` and `getattrlist` together comprising close to 80% of the access types for most workloads. In a majority of the iLife workloads, calls to `getattrlist` substantially exceed those to `stat`; however, `stat` calls hold a plurality in most iWork workloads, except the Pages tasks that deal with images. Data accesses are the third-most common category and generally occupy 10–20% of all accesses, peaking at 30% in iTunes PlayS. `statfs` calls, while uncommon, appear noticeably in all workloads except iTunes PlayS; at their largest, in iPhoto Del, they occupy close to 25% of the categories. Finally, `xattr` calls are the rarest, seldom comprising more than 3–4% of accesses in a workload, though still appearing in all tasks other than those in iTunes (in several cases, such as iPhoto Del, they comprise such a small portion of the overall workload that they do not appear in the plot).

Summary. As seen previous studies, metadata accesses are very common, greatly outnumbering accesses to file data across all of our workloads. `stat` is no longer as predominant as it has been, however; in many cases, `attrlist` calls appear more frequently. `statfs` and `xattr` access are not nearly as common, but still appear in almost all of the workloads. As there are usually at least five metadata accesses for each data

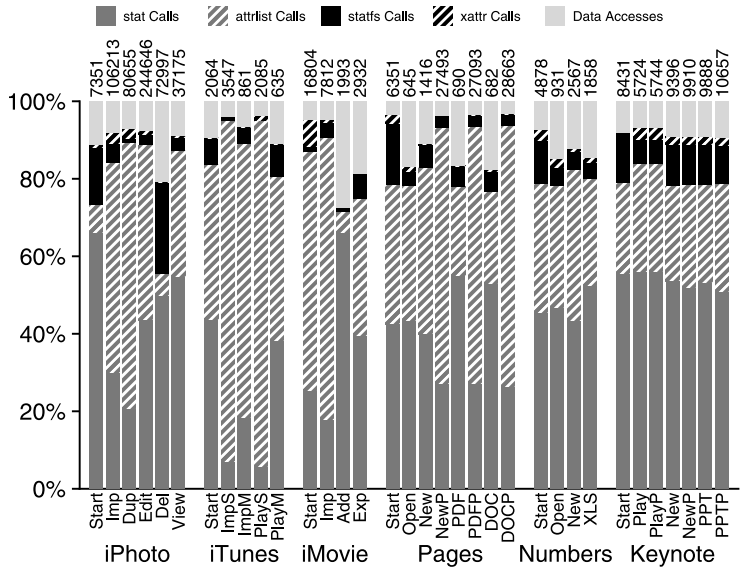

Fig. 20. Relative quantities of metadata access to data access. This plot shows the ratios of stat calls, statfs calls, and xattr accesses to data accesses in each task. The total combined amount of these is provided at the end of each bar.

access, the need to keep these system calls optimized, as described by Jacob et al. [2000], remains.

4.4. Memory: Caching, Buffering, and Prefetching

In this section, we explore application behavior that affects file-system memory management. We measure how often previously accessed data is read again in the future, how often applications overwrite previously written data, and how frequently hints are provided about future patterns. These measures have implications for caching, write buffering, and prefetching strategies.

4.4.1. Reuse and Overwrites. File systems use memory to cache data that may be read in the future and to buffer writes that have not yet been flushed to disk. Choosing which data to cache or buffer will affect performance; in particular, we want to cache data that will be reused, and we want to buffer writes that will soon be invalidated by overwrites. In this section, we measure how often tasks reuse and overwrite data. We keep track of which regions of files are accessed at byte granularity.

Figure 21 shows what portion of all reads are to data that was previously accessed. For most tasks, 75–100% of reads are to fresh data, but for eight tasks, about half or more of the data was previously accessed. The previously accessed data across all tasks can be fairly evenly divided into *previously read* and *previously written* categories.

Several of the tasks that exhibit the read-after-write pattern are importing data to an application library (i.e., iPhoto Imp, iTunes ImpS, and iMovie Imp). For example, iPhoto Import first performs reads and writes to copy photos to a designated library directory. The freshly created library copies are then read in order to generate thumbnails. It would be more efficient to read the original images once and create the library copies and thumbnails simultaneously. Unfortunately, use of two independent high-level abstractions, *copy-to-library* and *generate-thumbnail*, cause additional I/O.

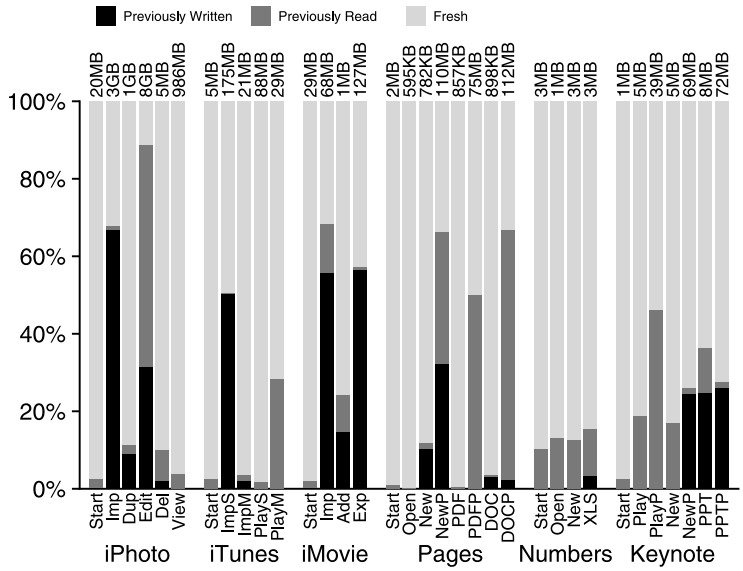

Fig. 21. Reads of previously accessed data. This plots shows what portion of reads are to file regions that were previously read or written. The numbers atop the bars represent all the bytes read by the tasks. The bars are broken down to show how much of this read data was previously read or written. Data previously read and written is simply counted as previously written.

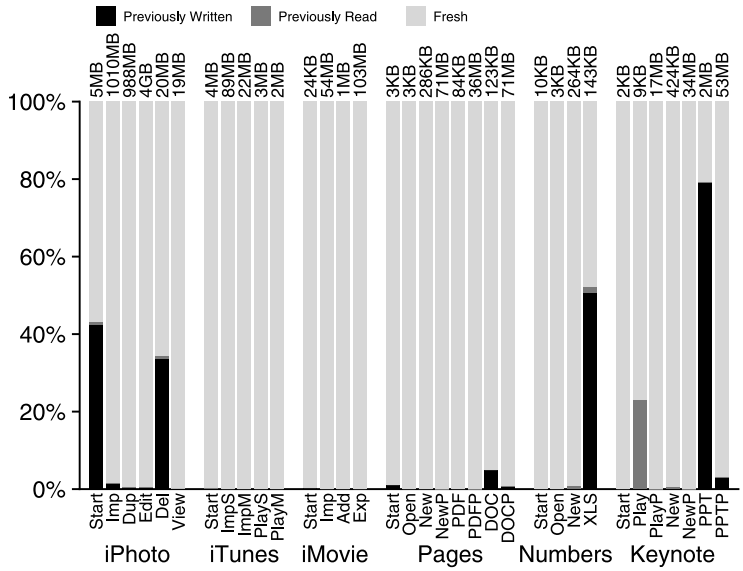

Fig. 22. Overwrites of previously accessed data. This plots shows what portion of writes are to file regions that were previously read or written. The numbers atop the bars represent all the bytes written by the tasks. The bars are broken down to show how many of these writes overwrite data that was previously read or written. Data previously read and written is simply counted as previously written.

Figure 22 shows what portion of all writes are overwriting data that has previously been read or written. We see that such overwrites are generally very rare, with four exceptions: iPhoto Start, iPhoto Delete, Numbers XLS, and Keynote PPT.

For the iPhoto tasks, the overwrites are caused by SQLite. For Numbers XLS and Keynote PPT, the overwrites are caused by an undocumented library, `SFCompatability`, which is used to export to Microsoft formats. When only a few bytes are written to a file using the `streamWrite` function of this library, the function reads a 512-byte page, updates it, and writes it back. We noted this behavior for Pages DOCP in the case study in Section 2; presumably the 512-byte granularity for the read-update-write operation is due to the Microsoft file format, which resembles a FAT file system with 512-byte pages. The read-update-write behavior occurs in all the tasks that export to Microsoft formats; however, the repetitious writes are less frequent for Pages. Also, when images are in the exported file, writes to the image sections dwarf the small repetitious writes. Thus, the repetitious write behavior caused by `SFCompatability` is only pronounced for Numbers XLS and Keynote PPT.

Summary. A moderate amount of reads could potentially be serviced by a cache, but most reads are to fresh data, so techniques, such as intelligent disk allocation, are necessary to guarantee quick access to uncached data. Written data is rarely overwritten, so waiting to flush buffers until data becomes irrelevant is usually not helpful. Many of the reads and writes to previously accessed data which do occur are due to I/O libraries and high-level abstractions.

4.4.2. Caching Hints. Accurate caching and prefetching behavior can significantly affect the performance of a file system, improving access times dramatically for files that are accessed repeatedly. Conversely, if a file will only be accessed once, caching data wastes memory that could be better allocated. Correctly determining the appropriate behavior can be difficult for the file system without domain-specific knowledge. Thus, Mac OS X allows developers to affect the caching behavior of the file system through two commands associated with the `fcntl` system call, `F_NOCACHE` and `F_RDADVISE`. `F_NOCACHE` allows developers to explicitly disable and enable caching for certain file descriptors, which is useful if the developer knows that either all or a portion of the file will not be reread. `F_RDADVISE` suggests an asynchronous read to prefetch data from the file into the page cache. These commands are only helpful, however, if developers make active use of them, so we analyze the frequency with which they appear in our traces.

Figure 23 displays the percent of file descriptors with `F_RDADVISE` issued, `F_NOCACHE` enabled, and `F_NOCACHE` both enabled and disabled during their lifetimes. The figure also includes opened file descriptors which received no I/O, even though most of our plots exclude them (sometimes files are opened just so an `F_RDADVISE` can be issued). We observed no file descriptors where `F_NOCACHE` was combined with `F_RDADVISE`. Overall, we see these commands used most heavily in iPhoto and iTunes. In particular, over half of the file descriptors opened on files in iPhoto Dup and Edit receive one of these commands, with `F_NOCACHE` overwhelmingly dominating Dup and `F_RDADVISE` dominating Edit. Most of the other iLife tasks tend to use `F_NOCACHE` much more frequently than `F_RDADVISE`, with the exception of iPhoto Start and Del. In contrast, only the Start workloads of the iLife applications issue any of these commands to more than one or two percent of their file descriptors, with `F_RDADVISE` occurring most frequently. When `F_NOCACHE` occurs (usually in those applications dealing with photos), it is usually disabled before the file descriptor is closed.

To complement this, we also examine the relative number of inodes in each workload that receive these commands. Figure 24 shows our results. `F_RDADVISE` is generally issued to a much smaller proportion of the total inodes than total file descriptors, indicating that advisory reads are repeatedly issued to a small set of inodes in the workloads in which they appear. In contrast, the proportion of inodes

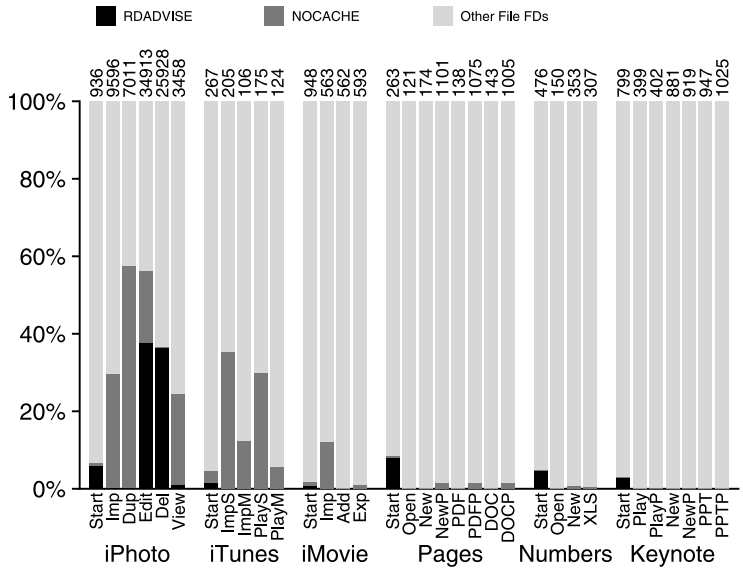

Fig. 23. File descriptors given caching commands. This plot shows the percent of file descriptors issued fcntl system calls with the F_RDADVISE and F_NOCACHE commands, distinguishing between file descriptors that eventually have F_NOCACHE disabled and those that only have it enabled. The numbers atop the bars indicate the total number of file descriptors opened on files.

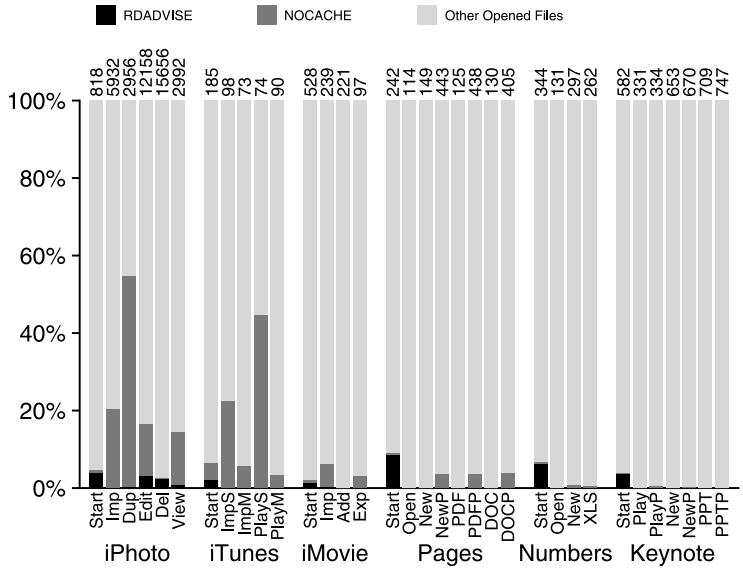

Fig. 24. Inodes affected by caching commands. This plot shows the percent of the inodes each task touches that are opened by file descriptors with either F_RDADVISE or F_NOCACHE. The numbers atop the bars indicate the total inodes touched by each task.

affected by F_NOCACHE is generally comparable to (or, in the case of iTunes PlayS, greater than) the proportion of file descriptors, which shows that these inodes are usually only opened once.

Summary. Mac OS X allows developers to guide the file system's caching behavior using the `fcntl` commands `F_NOCACHE` and `F_RDADVISE`. Only iPhoto and iTunes make significant use of them, though all of the iBench applications use them in at least some of their tasks. When the commands are used, they generally occur in ways that make sense: files with caching disabled tend not to be opened more than once, whereas files that receive advisory reads are repeatedly opened. Thus, developers are able to make effective use of these primitives when they choose to do so.

4.5. Transactional Properties

In this section, we explore the degree to which the iBench tasks require transactional properties from the underlying file and storage system. In particular, we investigate the extent to which applications require writes to be durable; that is, how frequently they invoke calls to `fsync` and which APIs perform these calls. We also investigate the atomicity requirements of the applications, whether from renaming files or exchanging inodes. Finally, we explore how applications use file locking to achieve isolation.

4.5.1. Durability. Writes typically involve a trade-off between performance and durability. Applications that require write operations to complete quickly can write data to the file system's main memory buffers, which are lazily copied to the underlying storage system at a subsequent convenient time. Buffering writes in main memory has a wide range of performance advantages: writes to the same block may be coalesced, writes to files that are later deleted need not be performed, and random writes can be more efficiently scheduled.

On the other hand, applications that rely on durable writes can flush written data to the underlying storage layer with the `fsync` system call. The frequency of `fsync` calls and the number of bytes they synchronize directly affect performance: if `fsync` appears often and flushes only several bytes, then performance will suffer. Therefore, we investigate how modern applications use `fsync`.

Figure 25 shows the percentage of written data each task synchronizes with `fsync`. The graph further subdivides the source of the `fsync` activity into six categories. *SQLite* indicates that the SQLite database engine is responsible for calling `fsync`; *Archiving* indicates an archiving library frequently used when accessing ZIP formats; *Pref Sync* is the `PreferencesSynchronize` function call from the Cocoa library; *writeToFile* is the Cocoa call `writeToFile` with the `atomically` flag set; and finally, *FlushFork* is the Carbon `FSFlushFork` routine.

At the highest level, the figure indicates that half the tasks synchronize close to 100% of their written data while approximately two-thirds synchronize more than 60%. iLife tasks tend to synchronize many megabytes of data, while iWork tasks usually only synchronize tens of kilobytes (excluding tasks that handle images).

To delve into the APIs responsible for the `fsync` calls, we examine how each bar is subdivided. In iLife, the sources of `fsync` calls are quite varied: every category of API except for Archiving is represented in one of the tasks, and many of the tasks call multiple APIs which invoke `fsync`. In iWork, the sources are more consistent; the only sources are Pref Sync, SQLite, and Archiving (for manipulating compressed data).

Given that these tasks require durability for a significant percentage of their write traffic, we next investigate the frequency of `fsync` calls and how much data each individual call pushes to disk. Figure 26 groups `fsync` calls based on the amount of I/O performed on each file descriptor when `fsync` is called, and displays the relative percentage each category comprises of the total I/O.

These results show that iLife tasks call `fsync` frequently (from tens to thousands of times), while iWork tasks call `fsync` infrequently except when dealing with images. From these observations, we infer that calls to `fsync` are mostly associated with media.

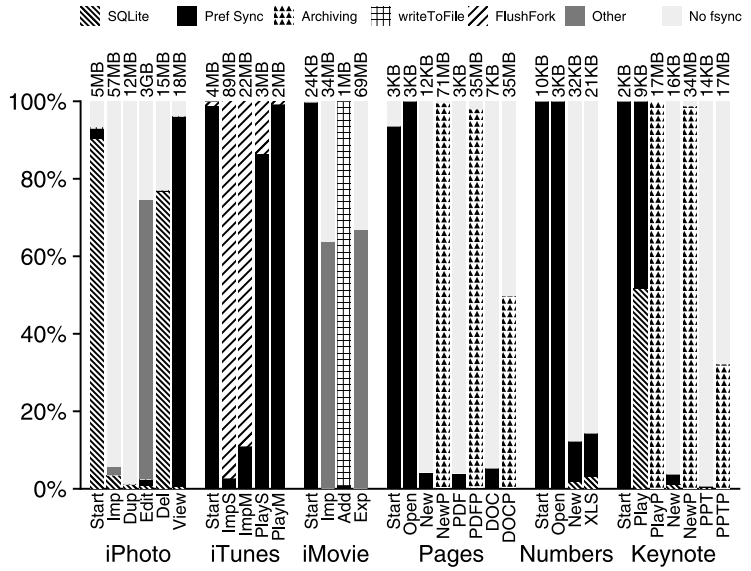

Fig. 25. Percentage of fsync bytes. The percentage of fsyncd bytes written to file descriptors is shown, broken down by cause. The value atop each bar shows total bytes synchronized.

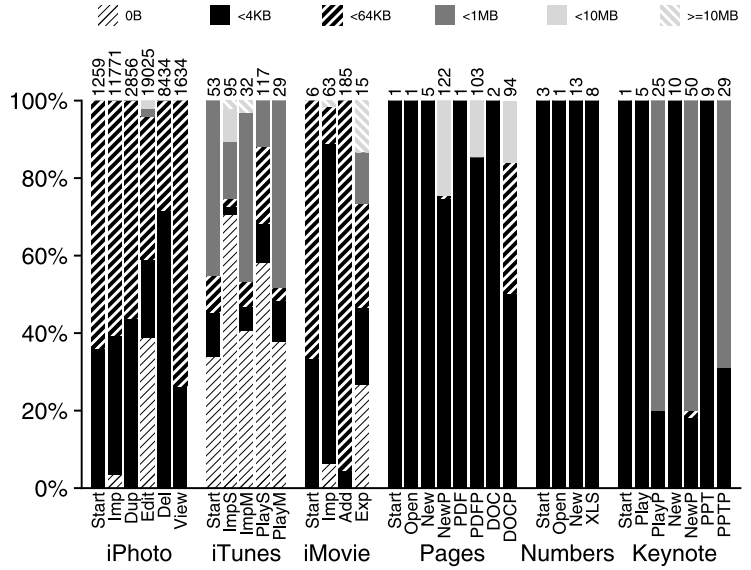

Fig. 26. Fsync sizes. This plot shows a distribution of fsync sizes. The total number of fsync calls appears at the end of the bars.

The majority of calls to `fsync` synchronize small amounts of data; only a few iLife tasks synchronize more than a megabyte of data in a single `fsync` call.

Summary. Developers want to ensure that data enters stable storage durably, and thus, these tasks synchronize a significant fraction of their data. Based on our analysis of the source of `fsync` calls, some calls may be incidental and an unintentional side-effect of the API (e.g., those from SQLite or Pref Sync), but most are performed

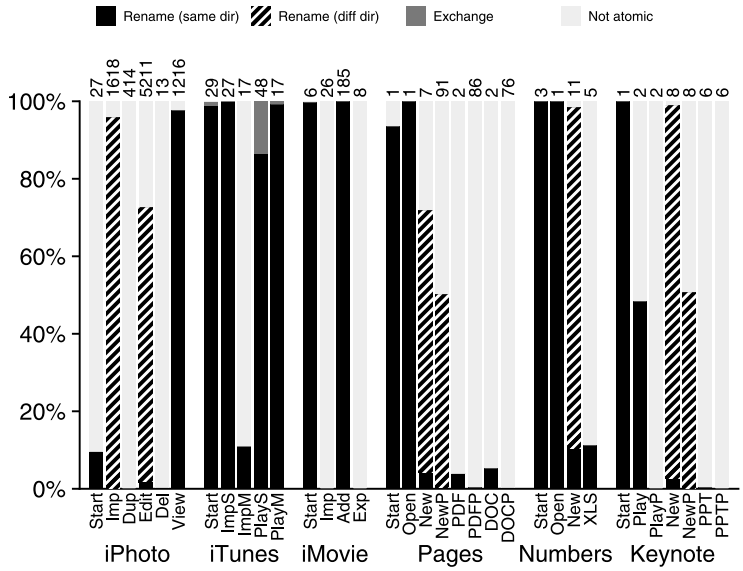

Fig. 27. Atomic writes. The portion of written bytes written atomically is shown, divided into groups: (1) rename leaving a file in the same directory; (2) rename causing a file to change directories; (3) exchangedata, which never causes a directory change. The atomic file-write count appears atop each bar.

intentionally by the programmer. Furthermore, some of the tasks synchronize small amounts of data frequently, presenting a challenge for file systems.

4.5.2. Atomic Writes. Applications often require file changes to be atomic. In this section, we quantify how frequently applications use different techniques to achieve atomicity. We also identify cases where performing writes atomically can interfere with directory locality optimizations by moving files from their original directories. Finally, we identify the causes of atomic writes.

Applications can atomically update a file by first writing the desired contents to a temporary file and then using either the `rename` or `exchangedata` call to atomically replace the old file with the new file. With `rename`, the new file is given the same name as the old, deleting the original and replacing it. With `exchangedata`, the inode numbers assigned to the old file and the temporary file are swapped, causing the old path to point to the new data; this allows the file path to remain associated with the original inode number, which is necessary for some applications.

Figure 27 shows how much write I/O is performed atomically with `rename` or `exchangedata`; `rename` calls are further subdivided into those which keep the file in the same directory and those which do not. The results show that atomic writes are quite popular and that, in many workloads, all the writes are atomic. The breakdown of each bar shows that `rename` is frequent; many of these calls move files between directories. `exchangedata` is rare and used only by iTunes for a small fraction of file updates.

We find that most of the `rename` calls causing directory changes occur when a file (e.g., a document or spreadsheet) is saved at the user's request. We suspect different directories are used so that users are not confused by seeing temporary files in their personal directories. Interestingly, atomic writes are performed when saving to Apple formats, but not when exporting to Microsoft formats. We suspect the interface

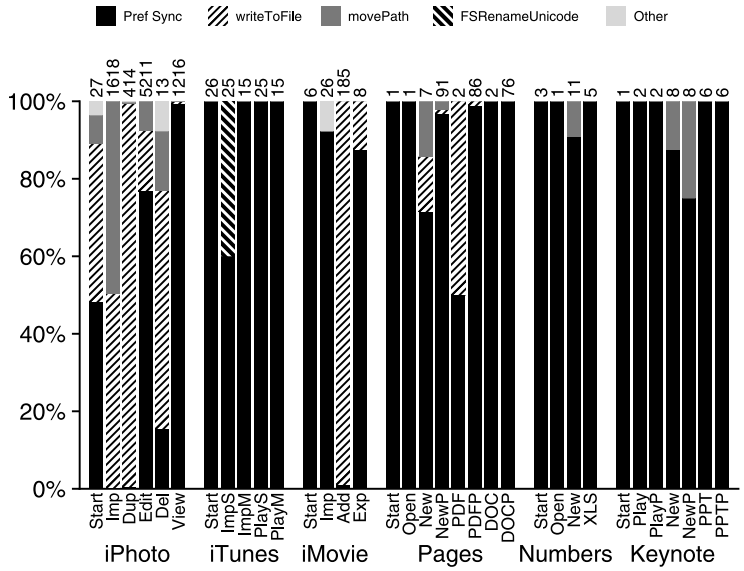

Fig. 28. Rename causes. This plot shows the portion of rename calls caused by each of the top four higher level functions used for atomic writes. The number of rename calls appears at the end of the bars.

between applications and the Microsoft libraries does not specify atomic operations well.

Figure 28 identifies the APIs responsible for atomic writes via `rename`. *Pref Sync*, from the Cocoa library, allows applications to save user and system wide settings in .plist files. *WriteToFile* and *movePath* are Cocoa routines and *FSRenameUnicode* is a Carbon routine. A solid majority of the atomic writes are caused by Pref Sync; this is an example of I/O behavior caused by the API rather than explicit programmer intention. The second most common atomic writer is writeToFile; in this case, the programmer is requesting atomicity but leaving the technique up to the library. Finally, in a small minority of cases, programmers perform atomic writes themselves by calling movePath or FSRenameUnicode, both of which are essentially `rename` wrappers.

Summary. Many of our tasks write data atomically, generally doing so by calling `rename`. The bulk of atomic writes result from API calls; while some of these hide the underlying nature of the write, others make it clear that they act atomically. Thus, developers desire atomicity for many operations, and file systems will need to either address the ensuing renames that accompany it or provide an alternative mechanism for it. In addition, the absence of atomic writes when writing to Microsoft formats highlights the inconsistencies that can result from the use of high level libraries.

4.5.3. Isolation Via File Locks. Concurrent I/O to the same file by multiple processes can yield unexpected results. For correctness, we need isolation between processes. Towards this end, UNIX file systems provide an advisory-locking API, which achieves mutual exclusion between processes that use the API. However, because the API is advisory, its use is optional, and processes are free to ignore its locks. The API supports both whole-file locking and file-region locking. File-region locking does not inherently correspond to byte regions in the file; instead, applications are free define their own semantics for the regions locked (e.g., a lock of size ten could cover ten records in the

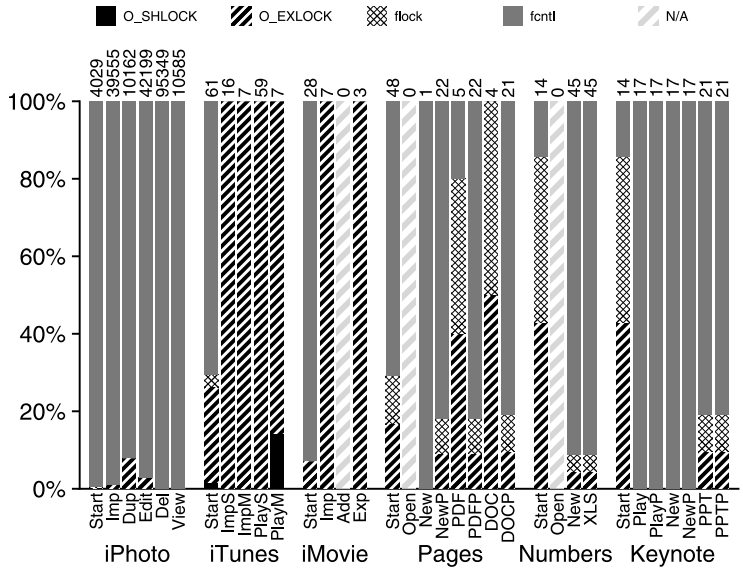

Fig. 29. Locking operations. The explicit calls to the locking API are shown, broken down by type. O_SHLOCK and O_EXLOCK represent calls to `open` with those flags, flock represents a change to a file's lock status via a call to `flock`, and fcntl represents file-region locking via `fcntl` with certain commands (`F_GETLK`, `F_SETLK`, or `F_SETLKW`). The number atop each bar indicates the number of locking calls.

file, each of which is 100 bytes long). We explore how the iBench applications use these locking API calls.

Figure 29 shows the frequency and type of explicit locking operations (implicit unlocks are performed when file descriptors are closed, but we do not count these). Most tasks perform 15-50 lock or unlock operations; only three tasks do not use file locks at all. iPhoto makes extreme use of locks; except for the Start task, all the iPhoto tasks make tens of thousands of calls through the locking API.

We observe that most calls are issued via the `fcntl` system call; these calls lock file regions. Whole-file locking is also used occasionally via the `O_SHLOCK` and `O_EXLOCK` `open` flags; the vast majority of these whole-file lock operations are exclusive. `flock` can be used to change the lock status on a file after it has been opened; these calls are less frequent, and they are only used to unlock a file that was already locked by a flag passed to `open`.

The extreme use of file-region locks by iPhoto is due to iPhoto's dependence on SQLite. Many database engines are server based; in such systems, the server can provide isolation by doing its own locking. In contrast, SQLite has no server, and multiple processes may concurrently access the same database files directly. Thus, file-system locking is a necessary part of the SQLite design [SQLite 2012].

Summary. Most tasks make some use of the locking API, and file-region locking accounts for the majority of this use. Tasks that heavily rely on SQLite involve numerous lock operations. In the case of iPhoto, it seems unlikely that other applications need to access iPhoto's database files; however, given SQLite's support for multi-process access, iPhoto must pay for the locking functionality, regardless. This shows that general-purpose storage APIs, such as SQLite, can result in unexpected inefficiency from actions like excessive locking.

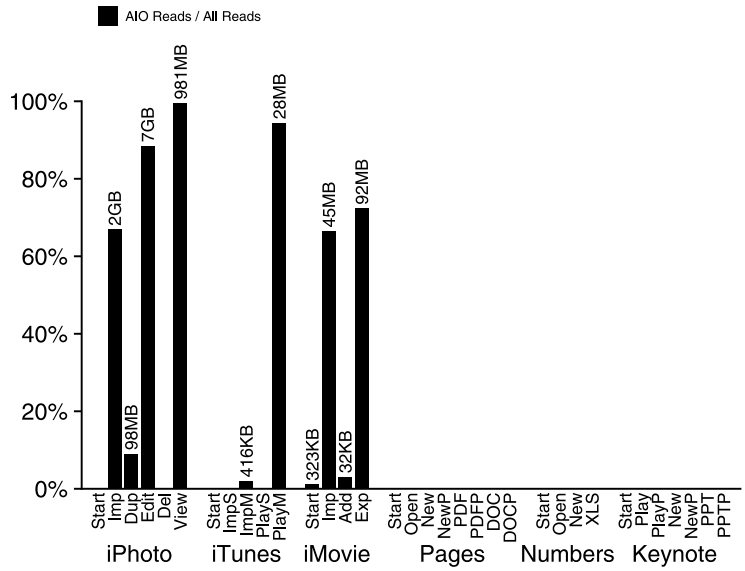

Fig. 30. Asynchronous reads. This plot shows the percentage of read bytes read asynchronously via `aio_read`. The total amount of asynchronous I/O is provided at the end of the bars.

4.6. Threads and Asynchronicity

Home-user applications are interactive and need to avoid blocking when I/O is performed. Asynchronous I/O and threads are often used to hide the latency of slow operations from users. For our final experiments, we investigate how often applications use asynchronous I/O libraries or multiple threads to avoid blocking.

Figure 30 shows the relative amount of read operations performed asynchronously with `aio_read`; none of the tasks use `aio_write`. We find that asynchronous I/O is used rarely and only by iLife applications. However, in those cases where asynchronous I/O is performed, it is used quite heavily.

Figure 31 investigates how threads are used by these tasks: specifically, the portion of I/O performed by each of the threads. The numbers at the tops of the bars report the number of threads performing I/O. iPhoto and iTunes leverage a significant number of threads for I/O, since many of their tasks are readily subdivided (e.g., importing 400 different photos). Only a handful of tasks perform all their I/O from a single thread. For most tasks, a small number of threads are responsible for the majority of I/O.

Figure 32 shows the responsibilities of each thread that performs I/O, where a thread can be responsible for reading, writing, or both. Significantly more threads are devoted to reading than to writing, with a fair number of threads responsible for both. This indicates that threads are the preferred technique for avoiding blocking and that applications may be particularly concerned with avoiding blocking due to reads.

Summary. Our results indicate that iBench tasks are concerned with hiding long-latency operations from interactive users and that threads are the preferred method for doing so. Virtually all of the applications we study issue I/O requests from multiple threads, and some launch I/Os from hundreds of different threads.

5. RELATED WORK

Although our study is unique in its focus on the I/O behavior of individual applications, a body of similar work exists both in the field of file systems and in

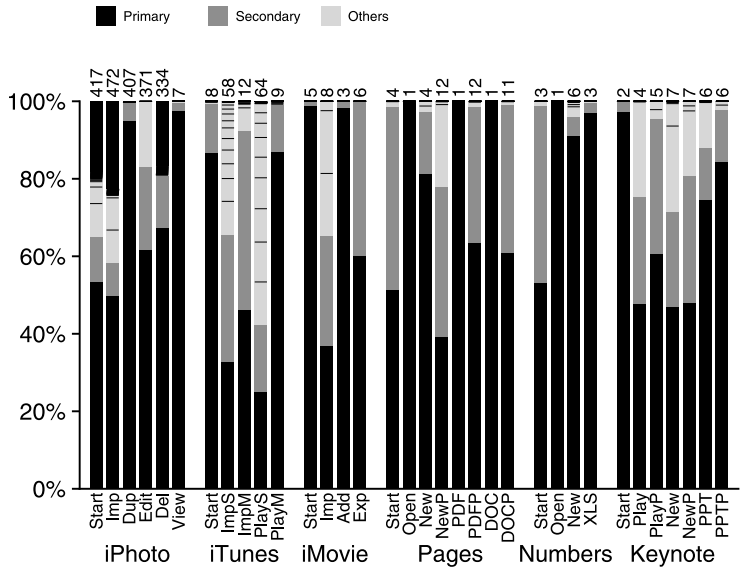

Fig. 31. I/O distribution among threads. The stacked bars indicate the percentage of total I/O performed by each thread. The I/O from the threads that do the most and second most I/O are dark and medium gray respectively, and the other threads are light gray. Black lines divide the I/O across the latter group; black areas appear when numerous threads do small amounts of I/O. The total number of threads that perform I/O is indicated next to the bars.

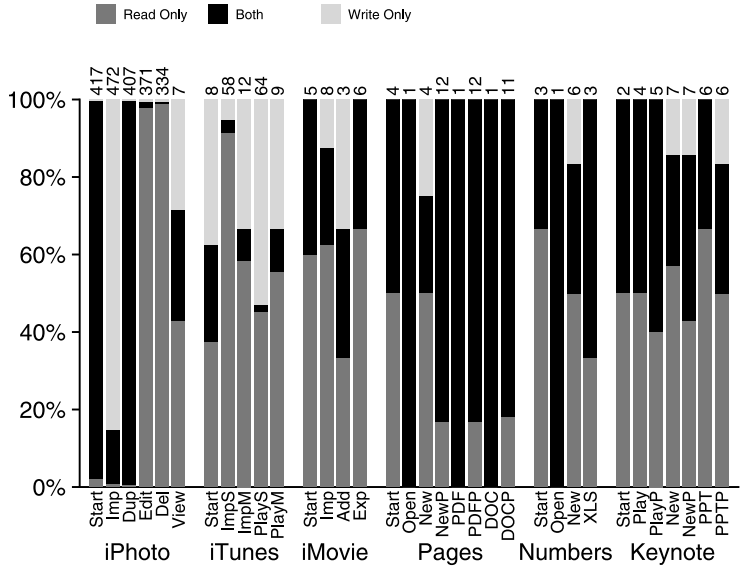

Fig. 32. Thread type distribution. The plot categorizes threads that do I/O into three groups: threads that read, threads that write, or threads that both read and write. The total number of threads that perform I/O is indicated next to the bars.

application studies. As mentioned earlier, our work builds upon that of Baker et al. [1991], Ousterhout et al. [1985], and Vogels [1999], and others who have conducted similar studies, providing an updated perspective on many of their findings.

However, the majority of these focus on academic and engineering environments, which are likely to have noticeably different application profiles from the home environment. Some studies, like those by Ramakrishnan et al. [1992] and by Vogels, have included office workloads on personal computers; these are likely to feature applications similar to those in iWork, but are still unlikely to contain analogues to iLife products. None of these studies, however, look at the characteristics of individual application behaviors; instead, they analyze trends seen in prolonged usage. Thus, our study complements the breadth of this research with a more focused examination, providing specific information on the causes of the behaviors we observe, and is one of the first to address the interaction of multimedia applications with the file system.

In addition to these studies of dynamic workloads, a variety of papers have examined the static characteristics of file systems, starting with Satyanarayanan's analysis of files at Carnegie-Mellon University [Satyanarayanan 1981]. One of the most recent of these examined metadata characteristics on desktops at Microsoft over a five year time span, providing insight into file-system usage characteristics in a setting similar to the home [Agrawal et al. 2007]. This type of analysis provides insight into long term characteristics of files that ours cannot; a similar study for home systems would, in conjunction with our article, provide a more complete image of how home applications interact with the file system.

While most file-system studies deal with aggregate workloads, our examination of application-specific behaviors has precedent in a number of hardware studies. In particular, Flautner et al.'s [2000] and Blake et al.'s [2010] studies of parallelism in desktop applications bear strong similarities to ours in the variety of applications they examine. In general, they use a broader set of applications, a difference that derives from the subjects studied. In particular, we select applications likely to produce interesting I/O behavior; many of the programs they use, like the video game Quake, are more likely to exercise threading than the file system. Finally it is worth noting that Blake et al. analyze Windows software using event tracing, which may prove a useful tool to conduct a similar application file-system study to ours in Windows.

6. DISCUSSION AND CONCLUSIONS

We have presented a detailed study of the I/O behavior of complex, modern applications. Through our measurements, we have discovered distinct differences between the tasks in the iBench suite and traditional workload studies. To conclude, we consider the possible effects of our findings on future file and storage systems.

We observed that many of the tasks in the iBench suite frequently force data to disk by invoking `fsync`, which has strong implications for file systems. Delayed writing has long been the basis of increasing file-system performance [Rosenblum and Ousterhout 1992], but it is of greatly decreased utility given small synchronous writes. Thus, more study is required to understand why the developers of these applications and frameworks are calling these routines so frequently. For example, is data being flushed to disk to ensure ordering between writes, safety in the face of power loss, or safety in the face of application crashes? Finding appropriate solutions depends upon the answers to these questions. One possibility is for file systems to expose new interfaces to enable applications to better express their exact needs and desires for durability, consistency, and atomicity. Another possibility is that new technologies, such as flash and other solid-state devices, will be a key solution, allowing writes to be buffered quickly, perhaps before being staged to disk or even the cloud.

The iBench tasks also illustrate that file systems are now being treated as repositories of highly-structured "databases" managed by the applications themselves. In some cases, data is stored in a literal database (e.g., iPhoto uses SQLite), but in most

cases, data is organized in complex directory hierarchies or within a single file (e.g., a .doc file is basically a mini-FAT file system). One option is that the file system could become more application-aware, tuned to understand important structures and to better allocate and access these structures on disk. For example, a smarter file system could improve its allocation and prefetching of "files" within a .doc file: seemingly nonsequential patterns in a complex file are easily deconstructed into accesses to metadata followed by streaming sequential access to data.

Our analysis also revealed the strong impact that frameworks and libraries have on I/O behavior. Traditionally, file systems have been designed at the level of the VFS interface, not breaking into the libraries themselves. However, it appears that file systems now need to take a more "vertical" approach and incorporate some of the functionality of modern libraries. This vertical approach hearkens back to the earliest days of file-system development when the developers of FFS modified standard libraries to buffer writes in block-sized chunks to avoid costly sub-block overheads [McKusick et al. 1984]. Future storage systems should further integrate with higher-level interfaces to gain deeper understanding of application desires.

Finally, modern applications are highly complex, containing millions of lines of code, divided over hundreds of source files and libraries, and written by many different programmers. As a result, their own behavior is increasingly inconsistent: along similar, but distinct code paths, different libraries are invoked with different transactional semantics. To simplify these applications, file systems could add higher-level interfaces, easing construction and unifying data representations. While the systems community has developed influential file-system concepts, little has been done to transition this class of improvements into the applications themselves. Database technology does support a certain class of applications quite well but is generally too heavyweight. Future storage systems should consider how to bridge the gap between the needs of current applications and the features low-level systems provide.

Our evaluation may raise more questions than it answers. To build better systems for the future, we believe that the research community must study applications that are important to real users. We believe the iBench task suite takes a first step in this direction and hope others in the community will continue along this path.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Rebecca Isaacs (our shepherd) for their tremendous feedback as well as members of our research group for their thoughts and comments on this work at various stages.

REFERENCES

AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. 2007. A five-year study of file-system metadata. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.

AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2009. Generating realistic impressions for file-system benchmarking. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.

APPLE COMPUTER, INC. 2011. AppleScript Language Guide. https://developer.apple.com/library/mac/documentation/applescript/conceptual/applescriptlangguide/AppleScriptLanguageGuide.pdf.

BAKER, M., HARTMAN, J., KUPFER, M., SHIRRIFF, K., AND OUSTERHOUT, J. 1991. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP'91)*. 198–212.

BARTLETT, W. AND SPAINHOWER, L. 2004. Commercial fault tolerance: A tale of two systems. *IEEE Trans. Depend. Secure Comput. 1*, 1, 87–96.

BLAKE, G., DRESLINSKI, R. G., MUDGE, T., AND FLAUTNER, K. 2010. Evolution of thread-level parallelism in desktop applications. *SIGARCH Comput. Archit. News 38*, 302–313.

BONWICK, J. AND MOORE, B. 2007. ZFS: The last word in file systems. http://opensolaris.org/os/community/ zfs/docs/zfs_last.pdf.

CANTRILL, B., SHAPIRO, M. W., AND LEVENTHAL, A. H. 2004. Dynamic instrumentation of production systems. In *Proceedings of USENIX'04*. 15–28.

DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'07)*.

DOUCEUR, J. R. AND BOLOSKY, W. J. 1999. A large-scale study of file-system contents. In *Proceedings of the SIGMETRICS'99*. 59–69.

ELLARD, D. AND SELTZER, M. I. 2003. New NFS tracing tools and techniques for system analysis. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)*. 73–85.

ENDO, Y., WANG, Z., CHEN, J. B., AND SELTZER, M. 1994. Using latency to evaluate interactive system performance. In *Proceedings of the USENIX 2nd Symposium on OS Design and Implementation (OSDI'96)*.

FLAUTNER, K., UHLIG, R., REINHARDT, S., AND MUDGE, T. 2000. Thread-level parallelism and interactive performance of desktop applications. *SIGPLAN Not. 35*, 129–138.

GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. 2003. The Google file system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'03)*. 29–43.

HAGMANN, R. 1987. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'87)*.

HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2011. A file is not a file: Understanding the I/O behavior of apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, 71–83.

HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1.

JACOB, D. R., LORCH, J. R., AND ANDERSON, T. E. 2000. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference*. 41–54.

LAMPSON, B. 1999. Computer systems research – Past and present. In *Proceedings of SOSP'17*.

LEE, E. K. AND THEKKATH, C. A. 1996. Petal: Distributed virtual disks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

LEUNG, A. W., PASUPATHY, S., GOODSON, G. R., AND MILLER, E. L. 2008. Measurement and analysis of large-scale network file system workloads. In *Proceedings of USENIX'08*. 213–226.

MACINTOSH BUSINESS UNIT (MICROSOFT). 2006. It's all in the numbers... blogs.msdn.com/b/macmojo/archive/2006/11/03/it-s-all-in-the-numbers.aspx.

MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2*, 3, 181–197.

MOGUL, J. C. 1994. A better update policy. In *Proceedings of USENIX Summer'94*.

OLSON, J. 2007. Enhance your apps with file system transactions. http://msdn.microsoft.com/enus/magazine/cc163388.aspx.

OUSTERHOUT, J. 1995. Why threads are a bad idea (for most purposes). www.standard.edu/class/cs240/readings/theads-bad-usenix96.pdf.

OUSTERHOUT, J. K., COSTA, H. D., HARRISON, D., KUNZE, J. A., KUPFER, M., AND THOMPSON, J. G. 1985. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'85)*. 15–24.

PATTERSON, D., GIBSON, G., AND KATZ, R. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD Conference (SIGMOD'88)*. 109–116.

PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. 1995. Informed prefetching and caching. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'95)*. 79–95.

PIKE, R. 2010. Another go at language design. http://www.stanford.edu/class/ee380/Abstracts/100428.html.

PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2005a. Analysis and evolution of journaling file systems. In *Proceedings of USENIX'05*. 105–120.

PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2005b. IRON file systems. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'95)*. 206–220.

RAMAKRISHNAN, K. K., BISWAS, P., AND KAREDLA, R. 1992. Analysis of file I/O traces in commercial computing environments. *SIGMETRICS Perform. Eval. Rev. 20*, 78–90.

RITCHIE, D. M. AND THOMPSON, K. 1973. The UNIX time-sharing system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'73)*.

ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. 2000. A comparison of file system workloads. In *Proceedings of USENIX'00*. 41–54.

ROSENBLUM, M. AND OUSTERHOUT, J. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1, 26–52.

SANDBERG, R. 1985. The design and implementation of the sun network file system. In *Proceedings of the USENIX Summer Technical Conference*. 119–130.

SATYANARAYANAN, M. 1981. A study of file sizes and functional lifetimes. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'81)*. 96–108.

SQLITE. 2012. SQLite: Frequently Asked Questions. http://www.sqlite.org/faq.html.

SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. 1996. Scalability in the XFS file system. In *Proceedings of USENIX'96*. San Diego, CA.

TILMANN, M. 2010. Apple's market share in the PC world continues to surge. maclife.com.

VOGELS, W. 1999. File system usage in Windows NT 4.0. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP'99)*. 93–109.

WOO, S. C., OHARA, M., TORRIE, E., SHINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture (ISCA'95)*. 24–36.

Received March 2012; accepted May 2012