TrapperKeeper: The Case for Using Virtualization to Add Type Awareness to File Systems

Daniel Peek
Facebook

Jason Flinn
University of Michigan

Abstract

TrapperKeeper is a system that enables the development of type aware file system functionality. In contrast to existing plug-in-based architectures that require a software developer to write and maintain code for each file type, TrapperKeeper requires *no* type-specific code. Instead, TrapperKeeper executes existing software applications that already parse the desired file type in virtual machines. It then uses accessibility APIs to control the application and extract desired information from the application's graphical user interface.

1 Introduction

Type awareness is increasingly important in modern storage systems. Traditionally, such systems managed files as a simple array of bytes; they were generally agnostic to the internal content of the files. However, many files have rich internal structure, such as ID3 headers of music files, and EXIF information in photos. Storage systems are quickly adding exciting new functionality based on understanding data internal to the files they store. For example, Apple's Spotlight tool [9], Windows Desktop Search [12], and Google Desktop [5] allow users to locate files based on internal metadata such as artist names in music files and comments in photo files. Graphical user interfaces (GUIs) can preview what documents would look like if they were opened in an application.

However, substantial development work is required to support this new functionality. The almost universal approach to understanding the internal structure of files is to require a software developer to write a plug-in that parses the file type and generates output for a general-purpose file system tool. For instance, given a new file type, a developer must write a plug-in to enable Spotlight to search through its metadata. She must write another parser to enable Google Desktop search for Windows, and yet another for Windows Desktop Search.

The required development effort is the root of several problems with the plug-in approach to type awareness:

- **It does not scale.** In total, developers must write a different parser for each file type, for each feature (e.g., preview and search), and for each file utility (e.g., Spotlight and Google Desktop). This development cost is incurred not only during the initial creation of plug-ins, but also during the evolution of that file type.
- **It inhibits innovation.** If an organization holds a dominant position in markets such as operating systems or search, then that organization can pressure developers to write plug-ins. Developers will acquiesce since they want their applications to work correctly for the majority of their users. But, innovators who do not enjoy such leverage struggle to convince developers to write plug-ins for systems that currently have small market shares.
- **It ignores the long tail.** While the most common file types may account for the majority of the files on a computer, the distribution of file types has a long tail. This means that even well-funded organizations willing to invest substantial resources may find it difficult to cover a very large percentage of files on a typical computer. We analyzed the file system snapshots from 8,729 desktops at Microsoft from the most recent year (2004) collected by Agrawal et al. [1] and found that the median file system has 603 distinct file name extensions and the distribution of file name extensions has a very long tail. Although file name extensions and file types are not necessarily in one-to-one correspondence, if one were to write type-specific code for each extension, even writing 50 plug-ins would not cover 24% of the files in the study. While it may be economically feasible to write plug-ins for the most popular file types, rarer file types will necessarily be unsupported. This will create a disruption for users of the new feature because some file types are unsearchable, do not have previews, etc.

In this paper, we present a solution to these problems, which we call TrapperKeeper. We observe that for a given file type, there likely exists an application that already understands how to parse, manipulate, and display

files of that type. Thus, there is no need to write separate plug-ins. Instead, TrapperKeeper uses virtualization to run such applications in isolation to extract specific features such as index terms or an image of a document being displayed. The goal is to eliminate work that must be done for file type to be supported. While this may impossible to satisfy fully, TrapperKeeper takes a large step toward achieving it.

2 Parsing with TrapperKeeper

TrapperKeeper has three components: Trapper, Keeper, and Grabber. Trapper is invoked once per file type to checkpoint an application inside a virtual machine just before it executes file parsing behavior. Later, Keeper is run when a new file of that type is added to the file system. It resumes the checkpoint and tricks the application within into parsing the new file. When parsing is complete, Grabber uses accessibility interfaces supplied by popular windowing systems to extract information from the application and present it to type aware features. For example, it might read the label and contents of a text field to generate a key/value pair that is stored in an indexing database. We have implemented an automatic extraction process for this data. However, it cannot deal with every possible interface. In the event that the automated process does not collect the right data, we have also built a mechanism to allow a human to demonstrate which data should be collected.

2.1 Trapper: Capturing application behavior

Trapper checkpoints an application just as it is about to execute its file parsing behavior but after it has completed its startup routines, displayed initial messages, shown open file dialogs, and so on. This checkpoint is later used by Keeper as a parser for file types associated with the application in the checkpoint.

To create the checkpoint, Trapper is given a VMWare virtual machine with the application to be captured installed. This virtual machine encapsulates the application, its dependencies, interactions with other processes, disk and memory state, and output. The application runs in as natural a situation as possible while still isolating all of the application's output and side effects. Trapper creates a shim file system in the guest OS that is used to detect when the application opens a file. The shim file system is implemented using FUSE [11] and appears to contain a single file, which we refer to as the *dummy file*.

Trapper uses the VMware VIX API to start the execution of the application within the virtual machine. Then, the application is directed to open the dummy file through the GUI or command line. The shim file system blocks the resulting `open` system call on the dummy file and checkpoints the virtual machine. It stores the resulting checkpoint of the application about to open the dummy file in a database of captured applications.

2.2 Keeper: Running application parsers

Keeper uses the application checkpoints produced by Trapper to parse individual files. Keeper induces the application to load a specific file and continue running from the checkpoint. The resulting GUI will be parsed and manipulated by feature-specific code using Grabber.

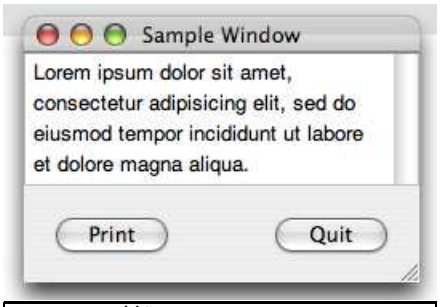
Given a file to parse, Keeper first determines which application checkpoint to use. By default, Keeper uses file extensions to determine the file type and select a corresponding application (e.g., files that end in ".mp3" are parsed using a checkpoint of the Exaile music player). However, Keeper is not limited by this assumption. Because Keeper activity has no side-effects, Keeper can try several parsers on a file, searching for one that parses the file correctly.

Keeper places the file to parse in a shared directory and resumes execution from the checkpoint taken by Trapper. In the checkpoint, the application was captured in the middle of an `open` system call that was being blocked by the shim file system. Keeper signals the shim file system to unblock the application and return from `open`. It redirects read-only operations on the dummy file to the file of interest instead. Write operations are buffered and returned to the application if it reads the same data that it wrote. No modifications are applied to the original file. This sleight-of-hand effectively replaces the contents of the dummy file with those of the file to parse. Thus, when the application proceeds, it blithely parses and displays the file.

2.3 Grabber: Capturing the interface

Grabber waits for the application to load the file and display its contents. Unfortunately, there is no standard method to detect when an application has finished parsing the file and its interface has reached a stable state in which it displays the file contents. A timeout-based approach is undesirable as applications may take varying amounts of time to load files. We also considered waiting until the GUI was unchanged for a fixed amount of time. However, this would not work for applications that do not have a stable final state due to activity such as playback of a media file.

Instead, we detect a final state by comparing the current state with the final state generated by the previous behavior of the same application. The first time Grabber uses an application checkpoint to parse a file, it waits a few minutes to be sure that the application GUI has reached a final state. Grabber then uses accessibility APIs to access a descriptive, hierarchical view of the elements of the GUI and writes it to a file. Figure 1 shows a simplified example of how a window and its child widgets appear when accessed through the accessibility API. The file generated by Grabber during this initial execution serves as an example of the final state of the application GUI after a successful parsing. During subsequent




```
Window - ‘‘Sample Window’’
   Close Button
   Minimize Button
   Zoom Button
   Button - ‘‘Print’’
   Button - ‘‘Quit’’
   Scroll Area
      Scroll Bar
      Text Area - ‘‘Lorem Ipsum...’’
```

Figure 1. Accessibility API view of a window

invocations, Grabber periodically reads the state of the application GUI through the accessibility API and computes the difference between the current state and the example state using the `diff` tool. Parsing is considered finished when the difference falls below a threshold (that decreases over time to guarantee termination).

Our method assumes that there should be a large difference between the final state of the application GUI and that in the example while the application is parsing the file. Once the file is loaded, some of the details of the GUI may be different from the example, but we expect much of the interface, such as the buttons, toolbars, and menu items, to be the same.

2.4 Discussion

Unlike the plug-in approach, TrapperKeeper does not need type-specific code to support a new file type. However, it does require support in the form of other software systems, most of which are readily available.

TrapperKeeper needs an application that can parse the file type in question and be directed to open a particular file. While this direction may require human effort once, the level of involvement is clearly much less than writing a plug-in. Further, the activity can be performed by any user of an application, not just by software developers.

The application must also implement an accessibility interface. Accessibility interfaces are part of every modern windowing system and are designed to support tools that assist visually impaired users. Most applications unknowingly implement an accessibility interface because they use the default GUI elements that come with the windowing system (buttons, windows, text fields, etc.) and those elements implement accessibility by default.

Finally, TrapperKeeper also needs an instance of the file type to let Grabber observe a successful parsing that establishes a baseline for comparison with future parsings.

3 Features

Once Grabber determines that the parsing application within the virtual machine has successfully reached its final GUI state, it executes feature-specific code that uses the interface state to gather information about the file that has just been parsed. Grabber implements functions that features can use to query and manipulate each application GUI. Besides reducing feature complexity by providing a common mechanism, this implementation allows us to abstract away specific implementation details of window managers and operating systems that may run within the virtual machine.

3.1 Metadata extraction

The metadata extraction feature produces attribute-value pairs for each file. It stores these pairs in a file indexing system. Since the attributes extracted by the feature are themselves type-specific, the file indexing tool is type-agnostic. For instance running the extraction feature on an MP3 file might produce the pairs {”artist”,”The Beatles”} and {”album”,”Revolver”}.

We have implemented two ways to select which metadata to extract. The first way is completely automatic. The metadata feature searches through the GUI information to find widgets that supply both names and values. For instance, a photo viewer might have a label widget with an attribute called “name” that has the value “location” and an attribute called “text” that has the value “Paris”. From this data, the metadata feature identifies {“location”,”Paris”} as an attribute-value pair. Besides labels, the automatic metadata extractor looks for text and tables. Tables are especially valuable as they often have column or row headers that describe cell contents.

The above method may not precisely extract the desired metadata, so we implemented an alternative, human-directed method for extracting metadata. The person who uses Trapper to create an application checkpoint subsequently runs Keeper on a sample file and selects the GUI widgets displaying metadata of interest by moving the mouse cursor over them and pressing a special key sequence (Ctrl-F11). The metadata feature makes a list of selected widgets. It extracts metadata from only those widgets in subsequent parsings. Selections are made once per file type. It does not require any programming skill, just the ability to use the application.

In many applications, a button or menu item can be used to display more detailed information about a file after it is opened. To access this information, we have added functionality that allows the user to specify buttons or menu items be activated when parsing is complete, but before metadata is extracted.

3.2 Document Preview

The second feature we implemented is document preview. This feature creates an image of the file being displayed by its parsing application; the image can then be used as an icon for that particular file by a graphical file browser.

The document preview feature uses Grabber to generate a screen shot of the application window after it has loaded a file. Grabber triggers the particular platform-specific screen shot functionality for the guest operating system running in the virtual machine. For better screenshots, the user can again specify GUI elements to be activated to perform actions such as switching software into presentation mode.

4 Evaluation

4.1 Executing features

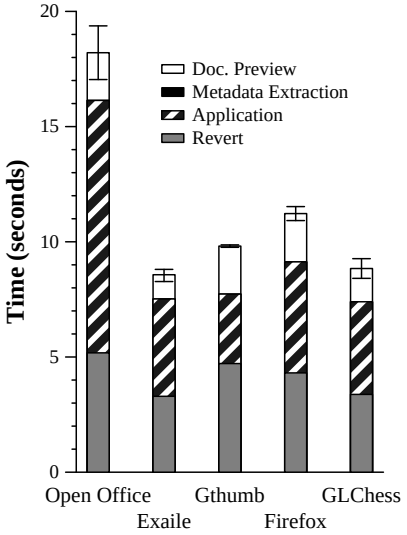
We measured the time to extract metadata from and generate document previews of five files. The first is a 1 MB Microsoft Word file that uses several fonts and includes several images; the document is opened by Open Office Writer. The second is a 4.9 MB MP3 music file opened by the Exaile media player. The remaining files are a 4.1 KB JPEG image, a 6.6 KB HTML file, and a 130 B saved chess game. These files are opened by the Gthumb image viewer, the Mozilla Firefox Web browser, and the GLChess chess program, respectively.

In these experiments, the computer we used was a Dell 690, with two quad-core 2.66 GHz Core 2 processors and 4 GB of RAM. The virtual machine we used was the 64-bit version of VMware workstation 6.0.2, and the host and guest OS were Ubuntu Linux 7.10. All experiments were done with a warm cache.

Figure 2 shows the time needed to resume from a virtual machine checkpoint and execute both features. As shown by the bottom segment of each bar, Keeper takes 3.3–5.2 seconds to resume the virtual machine from a Trapper checkpoint. Resuming from the Open Office Writer checkpoint takes slightly longer than the other applications, probably because Writer is more resource-intensive and uses more memory.

The second segment of each bar shows the time that Grabber waits for the application to reach a stable GUI state, an average of 5.4 seconds. Again, Open Office Writer takes the longest, 10.9 seconds, because it must convert and display a complex document. In contrast, if we resume Writer with a simple text document, a stable GUI state is reached in only 4.2 seconds. The difference, 6.7 seconds, shows the benefit of detecting a stable GUI state rather than using a fixed timeout. An algorithm with a fixed timeout would either wait too long for simple documents or not correctly capture complex ones.

The third segment of each bar, the execution of the metadata feature is almost instantaneous for all applications because Grabber dumps the state of the application



This figure shows time needed by Keeper and Grabber to perform metadata extraction and document preview on five files parsed by different applications. Results are the average of 5 trials and the error bars are 90% confidence intervals.

Figure 2. Keeper performance

GUI to determine that it has reached a stable state. Since the metadata feature needs identical information, Grabber can simply provide cached values. Parsing the metadata to extract attribute-value pairs takes negligible time.

The top segment of each bar shows the time for the document preview feature to take a screen shot of each application displaying its files, an average of 1.7 seconds.

Extrapolating from these results, TrapperKeeper could extract metadata and make previews of 318 files per hour. TrapperKeeper takes the longest amount of time, 18.2 seconds, to process the Word document. At this rate, it could still process 198 documents per hour.

Based on the file system study performed by Agrawal et al. [1], in 2004, the last year of their study of corporate desktops, the average file system contained approximately 90,000 files, 22% of which had been modified or created locally within the last year, a rate of only 2.26 per hour. The actual rate of is higher because the data does not capture files that are created and then deleted between file system snapshots. Nevertheless, the two orders of magnitude difference between the long-term file creation rate and TrapperKeeper's parsing rate gives us confidence that TrapperKeeper can keep up with the average user. Further, performance can be improved by using a faster checkpoint and rollback mechanism in the guest OS instead of reverting the entire virtual machine. Finally, TrapperKeeper can hybridize with existing plugin-based solutions, using fast plugins when they are available and TrapperKeeper when not, resulting in high performance for the most common types while enabling greater coverage with TrapperKeeper.

4.2 Reusing Applications

To demonstrate the general applicability of TrapperKeeper, we captured the type-specific behavior of every application listed in the applications menu on a fresh installation of Ubuntu 7.10. We restricted our experiment to the 20 listed applications that open user-specified files.

All but two of the applications worked with TrapperKeeper without complications. Of the remaining two, one simply required the play button to be pressed to make it `open` the file. The last required more trickery because it only opens files in a particular folder.

Overall, we found it quite easy to capture type-specific behavior with TrapperKeeper. Applying TrapperKeeper to all of these applications took a single person less than eight hours. Further, because many applications parse more than one file type, the 20 applications we handle allow us to support metadata extraction and document preview for over 100 distinct file types.

5 Related work

To the best of our knowledge, TrapperKeeper is the first project to leverage existing applications in order to extract metadata from files. This is in contrast to the technique first proposed by the Semantic File System [4], which uses special-purpose programs to extract metadata. This approach is used by today's popular metadata indexing systems. The plug-in approach has also been used in academic projects including Roma [10] and Stuff I've Seen [3]. More recently, document preview techniques [2] based on the same principles have emerged.

Activity put in context [6] also uses GUI information from existing applications to extract a different kind of metadata. It identifies files related to the user's current task by using which window is in front as a proxy for the active task.

TrapperKeeper makes use of accessibility APIs to get more extensive information from an application's GUI. DejaView [7] previously used accessibility APIs to archive and search the text displayed by applications. DejaView's purpose is similar to that of TrapperKeeper's metadata extraction feature. While DejaView indexes more than just file system data, TrapperKeeper's metadata extraction indexes data that the user has not viewed. It can also manipulate application GUIs through recorded actions like GUI scripting [8] to extract more information.

These are not the only ways to access information displayed to users. Screen scrapers have long been used to access information that programs display, but do not expose through other means. However, they are difficult to maintain as they require substantial custom code to extract the desired information and can easily be broken by changes in the application or its configuration.

6 Conclusion

In the past, unlocking the benefits of type awareness benefits has required software developers to build and maintain type-specific plug-ins. Since the cost of developing such plug-ins is high, even for the most popular features, many files will be unsupported because the distribution of file types has a long tail.

TrapperKeeper changes the economics of this equation by making it much easier to create type aware components. Any user of an application can create a Trapper checkpoint since no programming is required. Our results also show that TrapperKeeper can process hundreds of files per hour, a rate that far exceeds the amount of files created or modified by a typical user.

References

[1] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. In *FAST'07: Proceedings of the 5th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), USENIX Association, pp. 3–3.

[2] Quick look programming guide, April 2010. `http://developer.apple.com/documentation/UserExperience/Conceptual/Quicklook_Programming_Guide/Quicklook_Programming_Guide.pdf`.

[3] DUMAIS, S. T., E. CUTRELL, E., CADIZ, J. J., JANCKE, G., SARIN, R., AND ROBBINS, D. C. Stuff i've seen: A system for personal information retrieval and re-use. In *Proceedings of SIGIR 2003* (Toronto, Canada, 2003).

[4] GIFFORD, D. K., JOUVELOT, P., SHELDON, M. A., AND O'TOOLE, J. W. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pp. 16–25.

[5] Google desktop, April 2010. `http://desktop.google.com`.

[6] GYLLSTROM, K., SOULES, C., AND VEITCH, A. Activity put in context: identifying implicit task context within the user's document interaction. In *IIiX '08: Proceedings of the second international symposium on Information interaction in context* (New York, NY, USA, 2008), ACM, pp. 51–56.

[7] LAADAN, O., BARATTO, R., PHUNG, D., POTTER, S., AND NIEH, J. DejaView: A personal virtual computer recorder. In *Proceedings of the Twenty-first ACM Symposium on Operating Systems Principles* (Stevenson, WA, Oct 2007), pp. 279–292.

[8] LITTLE, G., LAU, T. A., CYPHER, A., LIN, J., HABER, E. M., AND KANDOGAN, E. Koala: capture, share, automate, personalize business processes on the web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2007), ACM, pp. 943–946.

[9] Spotlight overview, April 2010. `http://developer.apple.com/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.pdf`.

[10] SWIERK, E., KICIMAN, E., LAVIANO, V., AND BAKER, M. The Roma personal metadata service. In *Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications* (Monterey, CA, 2000).

[11] SZEREDI, M. Filesystem in userspace, April 2010. `http://fuse.sourceforge.net/`.

[12] Windows desktop home page, April 2010. `http://www.microsoft.com/windows/desktopsearch`.