TrapperKeeper: Using Virtualization to Add Type-Awareness to File Systems

Daniel Peek and Jason Flinn
{dpeek, jflinn}@eecs.umich.edu
Computer Science and Engineering
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 separate code modules for each new file type, TrapperKeeper requires *no type-specific code*. Instead, TrapperKeeper executes existing software applications that parse the desired file type inside virtual machines. It then uses accessibility APIs to control the application and extract desired information from the application's graphical user interface. We have implemented metadata extraction and document preview features that use TrapperKeeper, and we have used TrapperKeeper to capture the type-specific cognizance of over 20 applications that collectively parse more than 100 distinct file types. Our experimental results show that TrapperKeeper can execute these two features on hundreds of files per hour, a pace that far exceeds the rate that files are modified or created on the average desktop.

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, current file systems store more files that have rich internal structure, such as music files that have ID3 headers, photos that have EXIF information, and documents that contain formatting data and information about their pedigree. Storage systems are quickly adding exciting new functionality based on understanding data internal to the files they store. For example, Apple's Spotlight tool [21], Windows Desktop Search [26], and Google Desktop [13] 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) such as the Mac OS X Finder and Gnome display icons that preview what documents would look like if they were opened by applications that parse their particular file type.

However, much human effort 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. To enable document preview, she must write a different parser to generate a suitable image for files of that type.

As the above discussion highlights, there are several problems with the existing plug-in approach to adding type-awareness to file systems:

- **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). Thus, type-awareness has a large development cost; this cost is incurred not only during the initial creation of plug-ins, but also during the lifetime of that file type because developers must keep the plug-ins in sync with the applications that parse that file type (e.g., document preview should reflect the current appearance of documents in their corresponding applications).
- **It inhibits innovation.** If an organization holds a dominant position in a market such as operating systems or search, then that organization is in a position to dictate to developers that they must write plug-ins to support new features. Developers will generally acquiesce since they want their applications to work correctly for the majority of their users. But, innovators, who often do not enjoy a position of dominance in the marketplace, are left out in the cold. It is hard to convince developers to write plug-ins for operating systems or utilities that currently have small market shares, since the developers will only satisfy a small percentage of

their users in return for their hard work. If an innovator creates a new feature, such as a novel preview format, he will have a hard time convincing developers to support that feature. Thus, the innovator will usually find himself in the position of having to write plug-ins for most popular file types in order to bootstrap his innovation, even though he is probably unfamiliar with the details of those file types.

- **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 amounts of software developer time may find it difficult to cover a very large percentage of files on a typical computer. For instance, by analyzing the trace data collected by Agrawal et al. [2], we found that even if one were to write plug-ins to support the 50 most popular file extensions, 24% of the files observed during a large-scale study of corporate file system usage would still not have a corresponding plug-in. Thus, the total development effort required to write plug-ins for a new feature is quite large. While it may be economically feasible for a large organization to write and support plug-ins for a few of the most popular file types, rarer file types will necessarily be unsupported. This will create an intrusive disruption for users of the new feature because they must remember which file types are unsearchable, do not have previews, etc. when interacting with their computers.

In this paper, we present a solution to these problems, which we call TrapperKeeper. At the heart of our work is the observation that the application associated with a given file type 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 in order to extract specific features such as index terms or an image of a document being displayed.

TrapperKeeper has three components: Trapper, Keeper, and Grabber. Trapper is invoked once per file type. It creates a checkpoint of an application inside a virtual machine. By intercepting file system operations, Trapper takes the checkpoint at the exact moment the application opens a file of a specific type. Keeper is invoked when a new file of that type is added to the file system. It creates a virtual machine instance, and resumes it from the checkpoint. However, rather than return the original file requested by the application's `open` system call, Keeper supplies the file that was just added to the file system. Thus, the application within the virtual machine is tricked into opening the new file. Grabber uses accessibility interfaces supplied by popular windowing systems to extract information from the application. 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 file indexing and document preview features using TrapperKeeper. Our system requires no plug-ins or any other code to support a new file type for either feature. Our results show that Keeper and Grabber can extract metadata from and create previews for several hundred files per hour. This rate is far higher than necessary to support the amount of updates a typical user makes in a day. We have also used TrapperKeeper to capture the type-specific behavior of over 20 applications. Since many applications parse several different file types (e.g., a music player may parse MP3, WAV, AAC, and several other file types), TrapperKeeper currently supports over 100 distinct file types. Using TrapperKeeper, a single graduate student added metadata and preview support for all of these file types with less than eight hours of work.

We begin in the next section by discussing the goals we hope to achieve with the design of TrapperKeeper. Section 3 describes our implementation, and Section 4 details our file indexing and document preview features. Sections 5 and 6 describe our evaluation and discuss related work, respectively.

2 Design goals

Before we started on this project, we listed the most important properties for a type-awareness system to have. These properties differentiate TrapperKeeper from previous type-awareness systems.

2.1 Minimize work per file type

Ideally, we would like to allow new file types to benefit from features such as metadata indexing and document preview without requiring any additional work to support the new type. While this goal may be impossible to satisfy fully, we can take a large step toward achieving it by moving functionality out of type-specific components and into type-agnostic components, such as Trapper, Keeper, Grabber, and feature-specific code.

As shown in Figure 1, our design for TrapperKeeper has three tiers. The bottom tier, which consists of Trapper, Keeper, and Grabber, supplies generic functionality that is common across different features and file types. For instance, Trapper and Keeper checkpoint and resume virtual machines, while Grabber implements shared functionality to query and manipulate GUIs through their accessibility APIs.

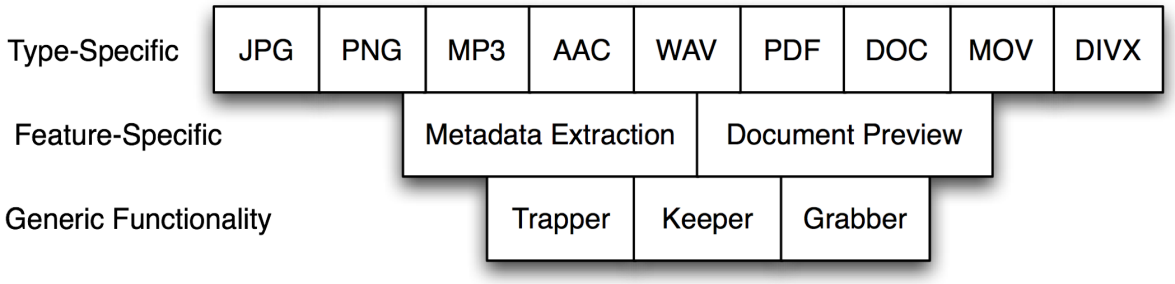


Figure 1. Tiers of increasing specificity

The second tier is feature-specific functionality. For instance, our metadata extraction feature uses Grabber to query the data in text fields and other GUI widgets. Our document preview feature uses Grabber to generate a screen snapshot of the application within the virtual machine. Functionality in this tier may have to be implemented for several different features, so we prefer to move any feature-agnostic code to the bottom tier. However, we expect the number of features to be quite small, say 10-20, compared to the number of file types, which we estimate to be several thousand based on our analysis of trace data in Section 5.6. Thus, it is reasonable to have a small amount of feature-specific functionality.

The top tier is type-specific functionality. Some examples of type-specific functionality are manipulating an application to generate the best screen shot for a document preview and selecting particular widgets that contain metadata to index. We wish to avoid implementing type-specific behavior whenever possible since such work is potentially multiplied by thousands of file types.

When we cannot avoid type-specific tasks, we aim to make them as easy as possible. For instance, TrapperKeeper can amortize the support for a single application across many distinct file types. Since TrapperKeeper operates on the application GUI, which is typically common for all file types, a single application snapshot can often be used for all types parsed by an application (e.g., for all image types accepted by an image viewer). In contrast, plug-in architectures operate on file data and thus require a separate plug-in for each distinct file type. Making type-specific tasks easier is also the motivation for our next design goal.

2.2 No code per file type

Current indexing tools such as Spotlight [21], Windows Desktop Search [26], and Google Desktop [13] rely on software developers to create and maintain plug-ins to parse files. Each plug-in is a piece of code that parses a particular file type, such as JPEGs or MP3s.

Although plug-ins are acceptable for the few most popular file types, there are a large number of less popular file types for which the benefit gained by adding them to the metadata system is outweighed by the development cost of the associated plug-in. Further, the different metadata systems require varying functionality from the plug-in, compounding the effort. For many file types, the original developers may be the only ones who know how to parse that file type. Yet, these developers may have gone out of business, lost interest, forgotten how to parse the file type, declared the file type obsolete, or may not have the engineering resources required to build plug-ins for each metadata system.

Metadata features that require the writing of new code for each file type will always run afoul of these problems. Thus, our approach is to *eliminate* all code specific to a file type. Instead, we run an application associated with each file type (e.g., Exaile for music files) and use the application to parse files of that type. When some human guidance is required, that guidance is provided through the application's GUI. A user runs an application using Trapper and clicks on widgets in the application's GUI to indicate which metadata should be indexed.

Our "no type-specific code" manifesto changes the economics of supporting file types. With TrapperKeeper, users with no programming expertise who are familiar with an application can add support for new file types simply by manipulating an application's GUI. When anyone can add support for a new type, even niche file types can be supported through the efforts of a single interested individual. Further, since a single user's efforts can be leveraged to add support for a type for other users, a participatory community can provide support for a wide variety of file types. This community has the potential for a much broader membership than, for example, the community of open-source developers because programming skill is not a prerequisite for membership.

2.3 Isolation

Our third design goal is to isolate applications used to parse files for features such as metadata extraction from the rest of the computer system. In plug-in-based systems, the plug-ins are typically run as part of a background process. Using applications to provide parsing behavior requires care because these applications often have complex, stateful interactions with other parts of the computing system such as the network, file system, and windowing system. A bug in a plug-in may cause other software to crash unless the plug-in is run in a sandbox. TrapperKeeper is potentially even more dangerous because it runs a complete copy of an application.

For this reason, TrapperKeeper provides strong isolation by running each parsing application in its own virtual machine. Virtualization prevents changes made within the virtual machine from being externalized to the host computer. Further, TrapperKeeper prevents buggy applications or malicious files designed to trigger bugs [6, 7] from affecting the parsing of subsequent files by reverting the virtual machine to a clean checkpoint after parsing each file.

3 Implementation

As shown in Figure 2, TrapperKeeper has three components that enable type-aware file system features. Trapper captures the file parsing behavior of an application and stores it for later use as an application-specific virtual machine checkpoint. Keeper resumes the virtual machine from the checkpoint to apply the captured application's behavior to a new file, generating a file-specific virtual machine. Grabber provides routines that allow features such as metadata extraction and document preview to examine and use the application interface within the virtual machine. We next describe each component.

3.1 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 to parse file types associated with the application in the checkpoint.

Trapper is only executed once for each application. The checkpoint it generates is used by all features that employ Keeper and Grabber. To create the checkpoint, Trapper is given a virtual machine that has the application to be captured installed. This virtual machine encapsulates the application, its dependencies, interactions

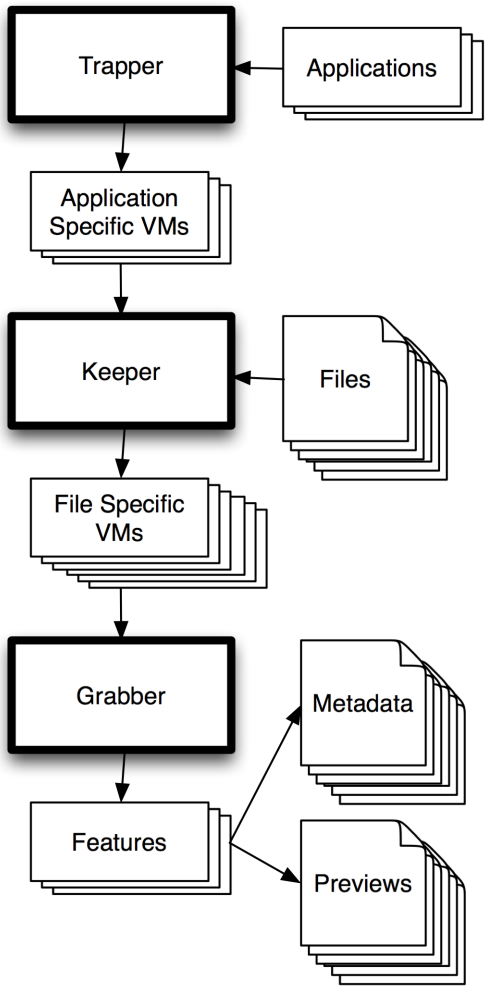


Figure 2. TrapperKeeper overview

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 to prevent them from becoming visible to applications running on the host computer. The virtual machine also constrains input to prevent misbehaving applications from accessing confidential data.

Virtual machine encapsulation makes the checkpoint captured by Trapper portable, which allows remote servers or workstations with spare resources to perform the parsing on behalf of other computers. This is especially convenient in the context of a distributed file system, in which all computers share a common namespace and can thus collaborate to extract type-specific features from new files. For mobile, resource-constrained clients such as cell phones, offloading parsing and feature execution to servers is especially attractive.

Trapper runs on the host operating system. Our current implementation uses VMware Workstation version

6.0.2 to execute the guest operating system and the parsing application. Trapper creates a file system that is shared between the host and guest operating system for communication. Trapper also 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 [10]. It appears to contain a single file, which we refer to as the *dummy file*.

Trapper is given the name of the application and any arguments on its command line. It uses the VMware VIX API to start the execution of the named application within the virtual machine. The VIX API provided by VMware Workstation allows applications on the host to invoke virtual machine features such as snapshot creation, virtual machine suspension, and running applications in the guest.

If the application can be directed to open the dummy file with a command-line argument, no further intervention is needed. Otherwise, the user must open the file using the application's GUI.

The shim file system detects and blocks the `open` system call on the dummy file and, in response, creates a file in the communication directory shared between the guest and host operating systems. Trapper checkpoints the virtual machine using the VIX API when this file appears. It stores the resulting checkpoint of the application about to open the dummy file in a database of captured applications.

3.2 Keeper: Running application parsers

Keeper uses the application checkpoints produced by Trapper to run feature-specific code on 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 the Grabber library. This is typically done whenever a new file enters the system or a file is modified.

Given a file to parse, Keeper must first determine which application checkpoint to use. By default, Keeper uses file extensions to determine the file type (e.g., files that end in ".mp3" are currently parsed using a checkpoint of the Exaile music player). While this default method adheres to the common practice of using file extensions to specify the type of each file, Keeper is not limited by this assumption. Because Keeper activity has no side-effects and the failure to parse a file can be detected, Keeper can try several parsers on a file, searching for one that parses the file correctly.

Keeper places the file of interest in the communication directory and uses the VIX API to resume 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. Through the shared file system, Keeper signals the shim file system to return from `open`. The shim file system unblocks the application and returns normally from `open`. However, instead of returning a file descriptor for the dummy file the application was opening at the moment the checkpoint was taken, the shim file system instead returns a FUSE handle that directs future operations for that file to the shim file system. The shim file system allows applications to open a file for both reading and writing. It applies read-only operations to the file of interest rather than the dummy file. Write operations are temporarily buffered and returned to the application if it reads the same data that it wrote. However, no modifications are ever applied to the original file. This sleight-of-hand replaces the contents of the dummy file with those of the file of interest. Thus, when the application proceeds, it blithely parses and displays the file of interest.

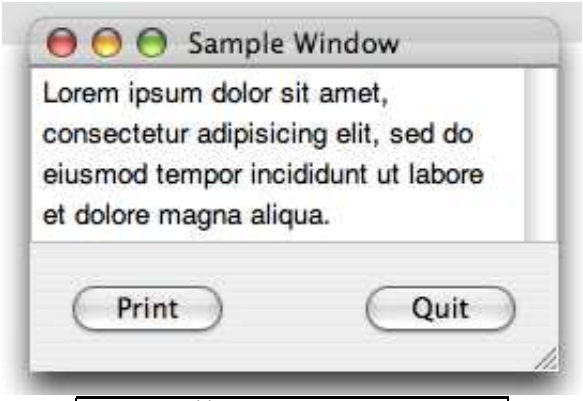
3.3 Grabber: Capturing the interface

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.

One possible solution would be for Grabber to wait a fixed amount of time after the file is opened. This solution is undesirable because choosing the right timeout is hard. If a timeout is chosen that is too small, features may read incorrect values from the display. If one is chosen that is too large, much time would be spent idle and the rate at which TrapperKeeper can process new files would be limited. Further complicating matters, applications may take varying amounts of time to load files depending on the size of the file being loaded or the current load on the host CPU and disk.

Another solution we considered was for Grabber to wait until no changes have been made to the GUI for a fixed amount of time. However, this approach would fail for applications that do not have a stable final state due to activity such as animated GUI elements or automatic playback of a media file.

The solution we chose detects 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. Accessibility APIs are a part




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

Figure 3. Accessibility API view of a window

of every modern windowing system and allow applications to examine the GUI elements of other processes and their properties such as displayed text and size. Grabber writes a representation of each GUI element of the display to a file, with each GUI element on a single line. Figure 3 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 the initial execution of the application checkpoint serves as an example of what the final state of the application GUI looks like after a successful parsing.

During subsequent invocations of the same application, Grabber periodically reads the state of the application GUI through the accessibility API. It then computes the difference between the current state and the example state. Our current implementation calculates the difference by counting the number of lines generated by the `diff` tool when given the current and example GUI data. More sophisticated methods are possible, but we have found that the `diff` approach works well in practice.

Our method assumes that there should be a large difference between the current 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.

Grabber checks the state of the interface every 100 ms. If the fraction of lines that differ is above a threshold, Grabber does not consider the interface to be in a final state and it continues to wait. Grabber starts the threshold at 40% and gradually increases it by 0.2% every 100 ms to guarantee termination. When the parsing and display is complete, Grabber invokes feature-specific code that uses the current GUI state to implement features such as metadata extraction and document preview.

3.4 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.

First, TrapperKeeper needs an application that can parse the file type in question. Ideally, the application should be able to open a file specified by a command-line argument. If this is not the case, a user must open a particular file using the application's GUI once. While this does represent human effort, 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.

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. The application must implement an accessibility interface to provide the windowing system access to information about the elements that make up the interface. Accessibility APIs were originally designed to support tools that assist visually impaired users by exporting information exposed by application GUIs. They are a part of every popular modern windowing system.

There are several forces driving applications to implement the accessibility interface. First, the default GUI elements (buttons, windows, text fields, etc.) automatically implement accessibility functionality. Because many applications use these default widgets rather than implement their own, those applications support accessibility without any extra development effort. Implementing the accessibility interface is also desirable for its original purpose, allowing visually impaired people to use the application. Finally, implementing an accessibility solution has become a requirement to sell software to the United States government [24].

4 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 generic code that each feature can use to query and manipulate each

application's 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.

We have implemented two features, metadata extraction and document preview, which are described in Sections 4.1 and 4.2. Additional features can be implemented by hooking into the Grabber API. Section 4.3 describes ideas for additional features that we have yet to implement.

Note that feature extensibility does not recreate the original problem of requiring too many plug-ins. A developer only needs to write one feature; in contrast, she would have to write a plug-in for every application on each platform (because the plug-in interfaces are currently platform-dependent) to support the same feature.

4.1 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 may be type-agnostic. For instance running the extraction feature on an MP3 file might produce the pairs {"composer","Beethoven"} and {"rating","five stars"}.

The metadata feature uses Grabber to acquire a dump of all GUI widgets such as text fields, choice boxes, tables, etc. The feature then sifts through the GUI information to find attribute-value pairs that describe the file.

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 can gather more data than is strictly necessary, so we implemented an alternative 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 selected widgets in subsequent parsings. The selection is only done once per file type. It does not require any programming skills, 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 an option that allows the user to specify buttons and/or menu items to be activated when parsing is complete, but before features are executed. The user chooses the buttons or menu items by parsing a sample file and then pressing a special key sequence (Ctrl-F10) while the mouse is hovering over the element to be activated. These selections are saved for use on further parsings.

Once the metadata feature has extracted attribute-value pairs for a file, it stores them in an indexing database from which they can later be used as search terms when searching for particular files. Our implementation uses Beagle [5] to store metadata. With TrapperKeeper, Beagle's insertion and retrieval functionality is similar to that provided by Apple's Spotlight and Microsoft's Windows Desktop Search, except that no plug-ins are required to parse files and generate metadata.

As an added benefit, the metadata stored by TrapperKeeper is expressed in terms of what users see on the screen when using applications, instead of in terms specified by a plug-in developer. For example, a music player may display a "play time" value for each song while a plug-in may refer to the same value as "length." Users making queries are more likely to use the former term since they see it every time they use the music player.

4.2 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. Windows, Mac OS X, and Gnome all provide mechanisms to set a file's icon. These captured images can easily be used to provide custom icons for files that reflect the contents of that file when rendered.

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 use Ctrl-F10 to specify GUI elements to be activated to perform actions such as switching software into presentation mode.

4.3 Other possible features

Besides the two features that we have built, we see several other possible features that could use the TrapperKeeper infrastructure.

4.3.1 Format transcoding

Many applications that open files have the ability to print those files. On modern platforms, the interface used to direct an application to print is shared by most applications. Because of this standardization, it is easy to use the accessibility API to direct applications to print the current document to a file. By capturing the printed output, a TrapperKeeper print feature can effectively transcode documents into a common format such as PostScript.

Transcoding documents to a common format has several benefits. It serves as a mechanism to create archival versions of documents in file formats that may become obsolete. The print feature also produces a preview of the entire document. Unlike the document preview feature described in Section 4.2 that produces a single image or thumbnail, the preview produced by the print feature would be a complete, browsable, indexable, and searchable representation of the document content. Further, by producing content in a standard format such as PostScript or PDF, the print feature enables content to be browsed and searched by standard tools that manipulate the common format.

4.3.2 Building ontologies

Different applications often use different terms to refer to the same thing. For example, one music player may display an "author" field to describe the person who wrote the music, while another may use the term "composer". An ontology feature could use such application diversity to develop context-dependent lists of synonyms.

The ontology feature could run multiple applications that parse the same data type and then ask Grabber to dump each application's GUI. It could generate attribute-value pairs for each application using the same method employed by the metadata extraction feature. By examining which pairs have the same value but different attribute names in two different application GUIs, the ontology feature could establish candidate synonyms.

4.3.3 Matching files to applications

A file matching feature could identify which applications parse a file of unknown type. Sometimes, users encounter a file that is missing an identifying extension in its name or one that has an unidentified extension. In such circumstances, it is difficult to know which applications can view or manipulate that file.

A straightforward file matching feature could try to use several applications captured by Trapper to open the file. After Keeper resumes each application from its checkpoint, Grabber can detect when the application successfully parses the file (in which case the state of the application GUI will be similar to the example provided when Keeper first opened a file supported by the application) and when the application cannot parse the file.

5 Evaluation

Our evaluation of TrapperKeeper sought to answer the following questions:

- How fast can TrapperKeeper extract metadata?
- How fast can TrapperKeeper generate previews?
- Can TrapperKeeper handle a variety of applications?
- What is the typical distribution of file types?

5.1 Methodology

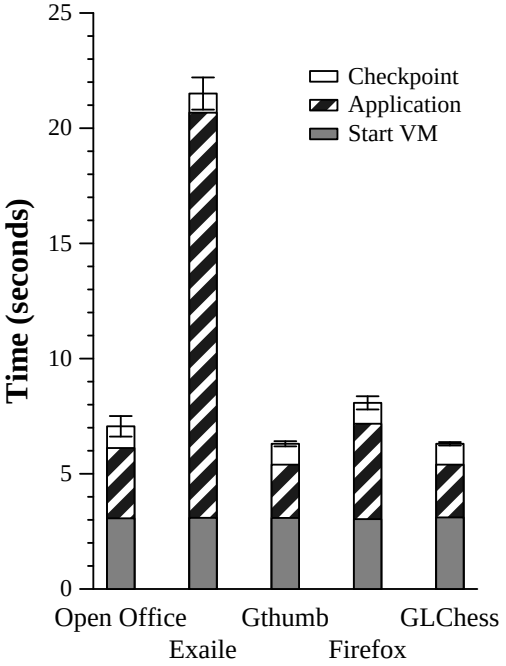
In the following experiments, the computer we used was a Dell 690, with two quad-core 2.66 GHz Core 2 processors and 4 GB of RAM. The virtual machine we used was the 64-bit version of VMware workstation 6.0.2, and the guest OS was Ubuntu Linux 7.10.

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. Exaile cannot be directed to open a file through its command line arguments, so we used Trapper to checkpoint the application after specifying the file to be opened using Exaile's GUI. 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; all three applications can open a file through command line arguments.

5.2 Trapper performance

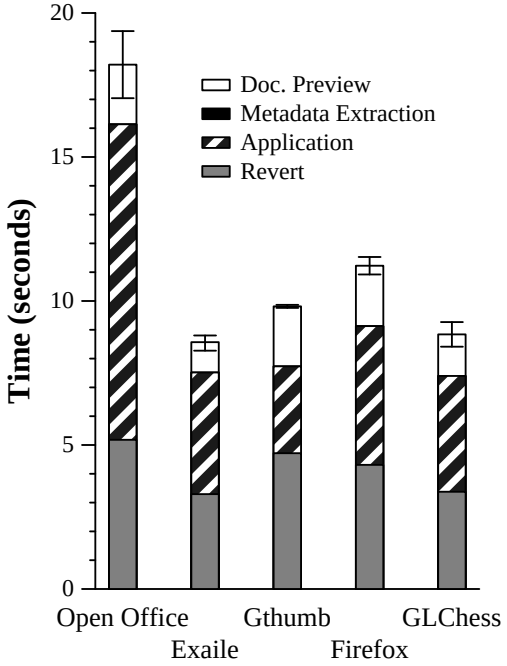
Figure 4 shows the time needed to create a checkpoint for each application. Creating a checkpoint for Exaile took the longest time because we had to manually open the MP3 file using Exaile's GUI. However, even with a manual step, it still took less than 22 seconds to create the checkpoint. Trapper created a checkpoint of all remaining applications in less than 9 seconds each.

We also measured the storage space consumed by checkpoints. The Ubuntu virtual machine took 4 GB. Each checkpoint consumed an additional 143-151 MB of storage. The checkpoints are relatively small because they are based on deltas from the virtual machine they were created from.



This figure shows the amount of time needed by Trapper to capture checkpoints of five applications. Each result is the average of 5 trials. The error bars show 90% confidence intervals.

Figure 4. Trapper performance



This figure shows the amount of time needed by Keeper and Grabber to execute two features, metadata extraction and document preview, on five files parsed by different applications. Each result is the average of 5 trials. The error bars show 90% confidence intervals.

Figure 5. Keeper performance

As the shadings within each bar in Figure 4 show, starting a new virtual machine took less than three seconds. Our initial checkpoint of Ubuntu Linux had all five applications installed. If any of our applications did not come with the base installation of the operating system, we would have had to install them before running Trapper. Checkpointing the virtual machine took 1-2 seconds, and the remaining time was spent waiting for the application to load and open the dummy file.

From these results, it is clear that creating a TrapperKeeper checkpoint is far less time-consuming than writing a type-specific plug-in.

5.3 Executing features

Figure 5 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 amount of time that Grabber waits for the application to reach a stable GUI state. Grabber waits an average of 5.4 seconds for each application. 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 between these two times, 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.

As shown by the third segment of each bar, the execution of the metadata feature is almost instantaneous for all applications. The reason is that Grabber must dump the state of each application's GUI to determine that the interface has reached a stable state. Since the metadata feature needs the identical information, Grabber can simply provide its cached values. Parsing the metadata to extract attribute-value pairs takes negligible time.

The top segment of each bar shows the amount of time for the document preview feature to capture a screen shot of each application displaying its files. Document preview takes an average of 1.7 seconds for all applications.

When the metadata extraction feature does not modify the state of the application (as is true for these 5 applications), TrapperKeeper executes the document preview feature immediately after the metadata extraction feature finishes. When this is not the case, TrapperKeeper must discard the virtual machine and again resume each appli-

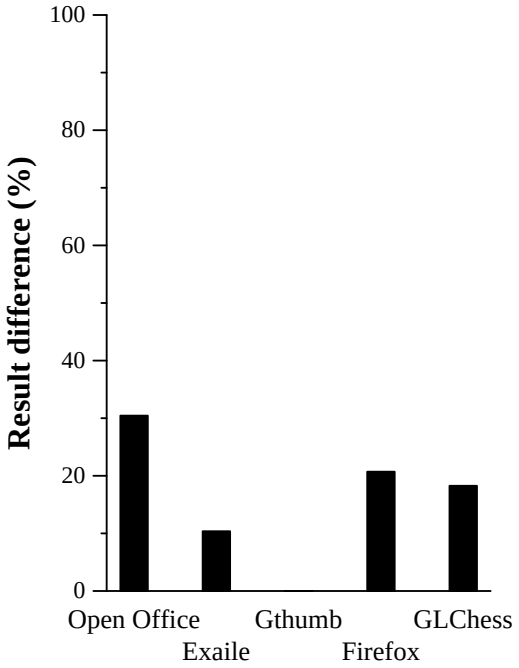


Figure 6. Difference in GUI states for different files

cation from its checkpoint prior to executing the document preview feature.

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.

According to a file system study performed by Agrawal et al. [2], 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 file creation is higher than this figure because the trace data does not capture files that are created and subsequently deleted between file system snapshots. Nevertheless, the more than 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 file creation rate of the average user. Further, all TrapperKeeper activity takes place asynchronously in the background, so it never blocks user file creation or modification.

Further, the increasing popularity of multicore computers on the desktop creates opportunities for parallelism. First, TrapperKeeper can parse new files on an idle core in the background while foreground applications use other cores. Additionally, because each TrapperKeeper application is encapsulated in a separate virtual machine, TrapperKeeper can be executed in parallel when more than one idle core is available. Since we had a multicore computer available to us, we verified this possibility by creating a multithreaded version of TrapperKeeper. We verified that two instances of TrapperKeeper could run in parallel and produce almost perfect speedup on 2 cores (e.g., approximately 395 documents per hour or 836 music files per hour). Beyond two cores, our implementation was limited by thread safety issues with the version of the VMware VIX API we used.

Further parallelism can be achieved in the context of a distributed file system. Keeper can run on any client of the distributed file system, leveraging the spare resources of multiple computers. This is made easy by the portability of virtual machines and distributed notification mechanisms such as persistent queries [18].

Performance can also be improved by hybridizing the TrapperKeeper approach with a plug-in based approach. By using plug-ins for a few of the most popular file types, such as MP3 and JPEG, we can make the these common cases fast for a small additional effort.

5.4 Detecting stable GUI states

We next verified that applications reach a stable GUI state with a relatively small number of differences from the initial example parsing (described in Section 3.3) when they open a different file of the same type.

To test this, we opened each application with a file of similar size but different contents than the one used by Grabber to create the example parsing. Rather than use Grabber's algorithm for detecting the final state, we let each application run and manually determined when the final state had been reached. Figure 6 shows the results. There is almost no difference in the final GUI state of Gthumb when it displays two different images because the image data is not part of the GUI state exported by the application, which makes sense since the original purpose of the accessibility APIs is to help visually impaired users. The GUI state of the remaining applications differ by 10.4–30.4%.

5.5 Experiences with TrapperKeeper

To demonstrate the general applicability of TrapperKeeper, we next 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 (not just configuration files).

All 20 applications were able to work with TrapperKeeper. However, not every application behaved as we expected. We discovered that choosing a minimal set of devices for the initial virtual machine is a bad idea.

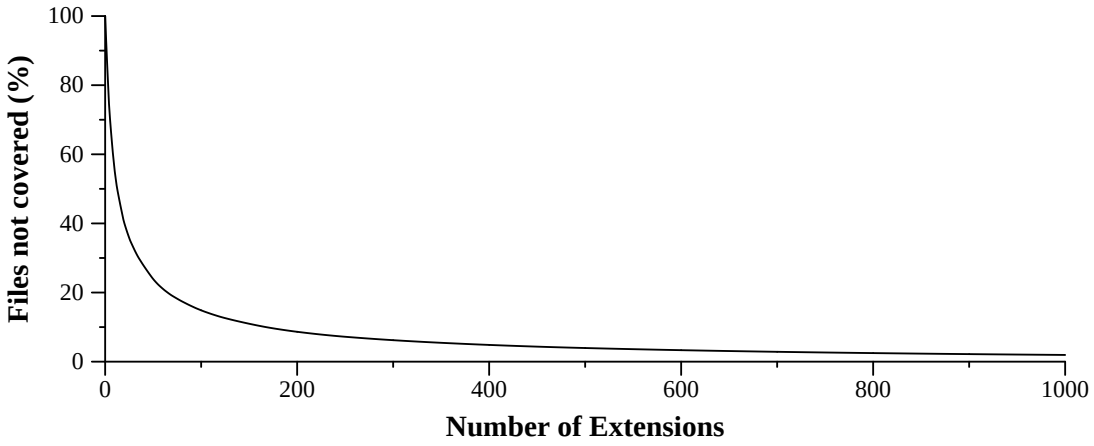


Figure 7. Fraction of files left uncovered

Two applications, Gnome Sound Recorder and Serpentine Audio CD Creator required audio devices and a CD writer, respectively. We corrected this problem by having the initial virtual machine come with a robust set of virtual devices.

For some applications, the actions required to execute the `open` system call are not obvious. For instance, Gnome Sound Recorder only records the name of files selected using its open file dialog. It does not execute the `open` system call until the play button is pressed. We handled such applications by experimenting with the GUI until an `open` system call was trapped by the shim file system.

Some applications were picky about file locations. Pidgin, an instant messenger client, only opens saved chat logs in a particular location in the file system, so we occupied that space with a symlink pointing to Trapper's dummy file. Fiddling was sometimes necessary to get an application's GUI to display file contents. For example, Evolution, an email client, requires several buttons to be pressed after opening a contact file before displaying its contents. We encountered one application, Exaile, that derives the type of a file from its filename extension. Since our dummy file can have only one name per application instance, we simply created snapshots with different dummy file names for each file type.

Despite these unexpected behaviors, we found it quite easy to capture type-specific behavior with TrapperKeeper. In fact, 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.

5.6 The long tail

One of our motivations for developing TrapperKeeper is that there are a large number of file types in use. Our hypothesis is that would take a Herculean effort to get near total coverage of file types using a plug-in approach. To get an idea of how many file types are in common usage, we examined the file system snapshots from the most recent year (2004) of the file system study performed by Agrawal et al. [2]. Their study captured file system snapshots on 8,729 desktop computers at Microsoft.

We used the methodology of Agrawal et al. to count the number of unique file name extensions and the percentage of files that appear for each extension. Their methodology defines a file name to have an extension if there are five or fewer characters following the last period in the file name. The endings of compressed files ending in .gz, .bz2, or .Z are removed before determining the file name extension. Using this methodology, approximately 7% of the file names captured by the study have no extension. In total, we found 102,869 distinct filename extensions in the study data. Although filename extensions and file types are not necessarily in one-to-one correspondence, this evidence certainly suggests that there are a large number of file types.

Figure 7 shows that the distribution of file name extensions has a very long tail. If one were to write type-specific code for each extension, even writing 50 plug-ins would not cover 23.92% of the files in the study. Table 8 shows the maximum number of files that would be covered by writing type-specific code for different numbers of extensions.

Because there are so many file types in the long tail, a strategy that involves substantial developer effort per file type will not achieve coverage. The cost of writing and maintaining code is high. Thus, it does not make

Number of Most Popular Extensions	Files Covered
10	44.94%
20	59.94%
50	76.08%
100	85.14%
1000	98.05%

Figure 8. Coverage of most popular file name extensions

economic sense to invest developer effort in supporting a file type whose percentage of the overall number of files is small. However, as our results show, although many individual file types are relatively unpopular, the collection of such unpopular types represents a substantial portion of the total number of files.

The solution that we propose is to reduce the effort needed to support each file type. TrapperKeeper requires no programming per file type. For other types, ordinary users can create snapshots using Trapper and an application's GUI. By greatly reducing the work required to support new file types, TrapperKeeper aims to increase the number of file types for which the benefit outweighs the cost of adding support.

6 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 [11], which uses small, special-purpose programs to extract metadata from files. The Semantic File System approach is used by today's popular metadata indexing and retrieval systems including Spotlight [21], Windows Desktop Search [26], Beagle [5], Google Desktop [13], and X1 [27]. For these commercial products, the special-purpose program is often a plug-in. The plug-in approach has also been used in academic projects including Roma [22] and Stuff I've Seen [8], which is the basis for Windows Desktop Search. More recently, document preview techniques [4] based on the same principles have emerged. Application developers provide plug-ins that are invoked to render a preview of the document.

TrapperKeeper makes use of accessibility APIs to get information from an application's GUI. These interfaces were designed to expose the structure and content of GUI elements to screen reader software, such as JAWS [9], VoiceOver [3], Narrator [1], and Gnopernicus [12], that help visually impaired users operate a computer. TrapperKeeper uses accessibility APIs for a different purpose: easy access to structured GUI information to extract information about displayed files.

DejaView [14] previously used accessibility APIs to archive and search the text displayed by applications. Thus, DejaView's purpose is similar to that of TrapperKeeper's metadata extraction feature. Both have different strengths. DejaView indexes more than just file system data. However, TrapperKeeper's metadata extraction feature indexes data that the user has yet to view. It also can manipulate application GUIs through recorded actions similar to GUI scripting [16] to extract more information.

More generally, interposing on calls between the application and windowing system has been used as a way to access and even modify [20] the user-visible aspects of applications.

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 generally undesirable as they require substantial custom code to extract the desired information and can easily be broken by changes in the application or its configuration.

TrapperKeeper also relies on a checkpoint and restart mechanism to capture application activity at the moment it begins to parse a file. This task is potentially difficult because we need to capture not only the application itself, but also its interactions with the windowing system and any processes or other entities with which the process communicates. Encapsulating the application in a virtual machine isolates the actions of the captured application from other applications and provides a convenient snapshot ability that captures the application at the moment a file is opened. Capturing an application in this way is similar to making a virtual machine appliance [25] whose only function is to parse files.

We chose to use virtual machine checkpoint and restart for TrapperKeeper. Potentially, we could have used a different checkpointing solution implemented in the operating system or at user level [15, 17, 19, 23]. Such a solution might provide better performance, but the implementation would be more challenging. For instance, special care would be needed to correctly handle the stateful interactions between the captured process and the window manager.

7 Conclusion

Understanding type-specific metadata encapsulated within files has a great many benefits. In the past, unlocking these benefits has required that software developers build and maintain type-specific plug-ins for a wide variety of features, tools, and operating systems. Since the cost of developing such plug-ins is high, it is hard to deploy innovative new features that exploit type-awareness. Even for the most popular features, many files on a computer will be unsupported because the distribution of file

types has a long tail and it is only possible to support the most popular file types.

TrapperKeeper changes the economics of this equation by making it much easier to create type-aware components. Our results show that checkpoints of new file types can be created in less than a minute using TrapperKeeper. Further, 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] Accessibility in Windows Vista. `http://www.microsoft.com/enable/products/windowsvista/`.

[2] 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.

[3] Apple - Mac OSX - VoiceOver. `http://www.apple.com/macosx/features/voiceover/`.

[4] Quick look programming guide. `http://developer.apple.com/documentation/UserExperience/Conceptual/Quicklook_Programming_Guide/Quicklook_Programming_Guide.pdf`.

[5] Main page - Beagle. `http://beagle-project.org`.

[6] CADAR, C., AND ENGLER, D. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software* (August 2005).

[7] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (2006).

[8] 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).

[9] JAWS for Windows. `http://www.freedomscientific.com/`.

[10] Filesystem in userspace. `http://fuse.sourceforge.net/`.

[11] 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.

[12] The gnome accessibility project. `http://developer.gnome.org/projects/gap/`.

[13] Google desktop. `http://desktop.google.com`.

[14] 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.

[15] LAADAN, O., AND NIEH, J. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *Proceedings of the 2007 USENIX Annual Technical Conference* (Santa Clara, CA, June 2007).

[16] 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.

[17] LITZKOW, M., AND SOLOMON, M. Supporting checkpointing and process migration outside the unix kernel. In *Usenix Winter Conference* (January 1992).

[18] PEEK, D., AND FLINN, J. EnsemBlue: Integrating consumer electronics and distributed storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, November 2006), pp. 219–232.

[19] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference* (January 1995), pp. 213–223.

[20] SATYANARAYANAN, M., FLINN, J., AND WALKER, K. R. Visual proxy: Expoiting OS customizations without application source code. *ACM SIGOPS Operating Systems Review 33*, 3 (1999), 14–18.

[21] Spotlight overview, May 2007. http://developer.apple.com/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.pdf.

[22] 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).

[23] THEIMER, M. M., LANTZ, K. A., AND CHERITON, D. R. Preemptable remote execution facilities for the V-System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Santa Clara, CA, December 1985).

[24] Section 508 of the rehabilitation act, as amended by the workforce investment act of 1998.

[25] Virtual appliance marketplace. http://www.vmware.com/appliances/.

[26] Windows desktop home page. http://www.microsoft.com/windows/desktopsearch.

[27] X1 - unified, actionable search. http://www.x1.com.