DejaView: A Personal Virtual Computer Recorder

Oren Laadan, Ricardo A. Baratto, Dan B. Phung, Shaya Potter, and Jason Nieh
{orenl, ricardo, phung, spotter, nieh}@cs.columbia.edu
Department of Computer Science
Columbia University

ABSTRACT

As users interact with the world and their peers through their computers, it is becoming important to archive and later search the information that they have *viewed*. We present DejaView, a personal virtual computer recorder that provides a complete record of a desktop computing experience that a user can playback, browse, search, and revive seamlessly. DejaView records visual output, checkpoints corresponding application and file system state, and captures displayed text with contextual information to index the record. A user can then browse and search the record for any visual information that has been displayed on the desktop, and revive and interact with the desktop computing state corresponding to any point in the record. DejaView combines display, operating system, and file system virtualization to provide its functionality transparently without any modifications to applications, window systems, or operating system kernels. We have implemented DejaView and evaluated its performance on real-world desktop applications. Our results demonstrate that DejaView can provide continuous low-overhead recording without any user noticeable performance degradation, and allows browsing, search and playback of records fast enough for interactive use.

Categories and Subject Descriptors: C.2.4 Computer-Communication-Networks: Distributed Systems–client/server; D.4.5 Operating Systems: Reliability–checkpoint/restart; D.4.3 Operating Systems: File Systems Management

General Terms: Design, Experimentation, Performance

Keywords: Desktop Search, Virtualization

1. INTRODUCTION

As users spend more time interacting with the world and their peers through their computers, it is becoming important to archive and later search the knowledge, ideas and information that they have seen through their computers. However, finding the information one has seen among the ever-increasing and chaotic sea of data available from a computer remains a challenge. Exponential improvements in processing, networking, and storage technologies are not making this problem easier. Computers are getting faster at generating, distributing, and storing vast amounts of data, yet humans are not getting any faster at processing it.

Some tools address aspects of this problem. Web search engines focus on static information available on the web, but do not help with a user's personal repository of data, dynamically generated and changing content created at the moment a user has viewed a web page, or hidden databases a user may have seen but are not available through web search engines [16]. Similarly, desktop file search tools return current files that may be of interest, but do not return results from files that are no longer available, or from information seen by the user but never actually saved to files.

Vannevar Bush's Memex vision [4] was to build a device that could store all of a user's documents and general information so that it could be quickly referenced. Building on that vision, we have created DejaView, a personal virtual computer recorder that provides a complete WYSIWYS (What You Search Is What You've Seen) record of a desktop computing experience. DejaView enables users to playback, browse, search, and revive records, making it easier to retrieve information they have seen before.

Leveraging continued exponential improvements in storage capacity [30], DejaView records what a user has seen as it was originally displayed with the same personal context and layout. All viewed information is recorded, be it an email, web page, document, program debugger output, or instant messaging session. DejaView enables a user to playback and browse records for information using functions similar to personal video recorders (PVR) such as pause, rewind, fast forward, and play. DejaView enables a user to search records for specific information to generate a set of matching screenshots, which act as portals for the user to gain full access to recorded information. DejaView enables a user to select a given point in time in the record from which to revive a live computing session that corresponds to the desktop state at that time. The user can time travel back and forth through what she has seen, and manipulate the information in the record using the original applications and computing environment.

DejaView transparently provides these features by introducing lightweight virtualization mechanisms and utilizing available accessibility interfaces. DejaView virtualizes the display to capture and log low-level display commands, enabling them to be replayed at full fidelity at a later time. It utilizes accessibility interfaces to simultaneously capture displayed text and contextual information to automatically index the display record so it can be searched. It combines display and operating system (OS) virtualization to decouple window system and application state from the underlying system, allowing them to be continuously checkpointed and later revived, while only saving user desktop state, not the entire OS instance. Checkpointing at this finer granularity, shifting expensive I/O operations out of the critical path, and using various optimizations such as fast incremental and copy-on-write techniques are crucial to minimize any impact on interactive desktop application performance. DejaView combines logging and unioning file system mechanisms to capture the file system state at each checkpoint. This ensures that applications revived from a checkpoint are given a consistent file system view corresponding to the time at which the checkpoint was taken.

DejaView's ability to browse and search display content and revive live execution provides a unique blend of functionality and performance. By browsing and searching the display record, the user is able to access content as it was originally seen, and quickly find information at much faster rates than if the information had to be generated by replaying execution. By reviving the execution environment, the user can go beyond a static display of content to fully manipulating and processing information using the same application tools available when the information was first displayed.

We have implemented DejaView as a set of loadable modules for Linux and the X Window System. It provide transparent operation without modifying, recompiling, or relinking applications, window systems, or OS kernels. To demonstrate its effectiveness, we have evaluated its performance on a wide-range of real-world desktop applications. Our results show that DejaView can provide continuous low-overhead recording without any user noticeable performance degradation of the system. Downtime due to checkpointing when running desktop application benchmarks is less than 10 ms, a time delay much shorter than what humans can readily detect [35]. Storage requirements of DejaView records at highest quality are comparable to PVRs in recording HDTV resolution media programming. As terabyte storage capacities become commonplace, DejaView enables high quality WYSIWYS recording to be used for everyday use. Our results also show that DejaView can provide much faster than real-time playback of records and supports browsing and searching of records fast enough for interactive use.

This paper presents the design and implementation of DejaView. Section 2 describes the DejaView usage model. Section 3 gives an overview of the DejaView architecture. Section 4 describes how display virtualization is used to record, index, playback, and search the visual output of the display. Section 5 describes how OS virtualization is used to enable fast continuous checkpointing of live user computing sessions and the ability to revive user sessions from any point in time. Section 6 discusses the implementation of our DejaView prototype in Linux and experimental results using real-world applications to evaluate the performance of DejaView. Section 7 discusses related work. Finally, we present some concluding remarks and directions for future work.

2. USAGE MODEL

DejaView operates transparently within a user's desktop, recording its state and indexing all text as the user interacts with the computer. The user can then later view the recorded session by playing it back and can interact with

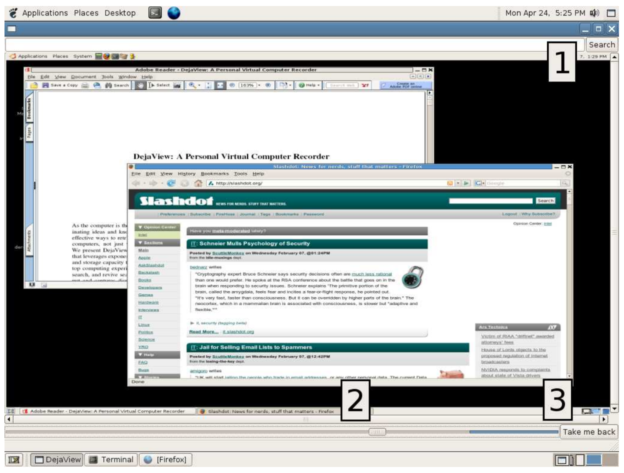
Applications Places Desktop
Mon Apr 24, 5:25 PM
Search
1
Adobe Reader - DejaView: A Personal Virtual Computer Recorder
DejaView: A Personal Virtual Computer Recorder
Slashdot: News for nerds, stuff that matters - Firefox
Slashdot
Schneier Mulls Psychology of Security
Jail for Selling Email Lists to Spammers
2
3
Take me back
DejaView
Terminal
[Firefox]

Figure 1: DejaView Screenshot showing widgets for playback and search inside a live desktop session. At the top right, the *Search* (1) button brings up a dialog to perform searches. At the bottom, the slider (2) allows the user to browse through the recording, and the *Take me back* (3) button revives the session at that point in time

any previous session state by reviving it. DejaView consists of a server that runs a user's desktop environment including the window system and all applications, and a viewer application. The viewer acts as a portal to access the desktop, sending mouse and keyboard events to the server which passes them to the applications. Similarly, screen updates are sent from the server to the viewer, which displays them to the user. This functional separation allows the viewer and server to run on the same or different computers.

The viewer provides three UI widgets to access DejaView's recording functionality, shown in Figure 1. A search button opens a dialog box to search for recorded information, with results displayed as a gallery of screenshots. A slider provides PVR-like functionality, allowing the user to rewind or fast-forward to different points in the record, or pause the display during live execution to view an item of interest. Finally, a *Take me back* button revives the desktop session at the point in time currently displayed.

DejaView users can choose to trade-off record quality versus storage consumption to meet their particular environment and needs. By default, display data is recorded at the original fidelity, but users can change the resolution and the frequency at which display updates are recorded. Application execution state is recorded according to a configurable policy that adjusts the rate of checkpointing based on display output and user input.

DejaView captures displayed text and associates it with visual output to index the display record for searching. Users can create additional annotations by simply typing anywhere on the screen, resulting in the automatic indexing of that text. Furthermore, DejaView allows the user to tag the current display state by typing text, selecting it with the mouse and pressing a combination key, to explicitly index the selected text with a special *annotation* attribute.

When the user revives a past session, an additional viewer window is used to access the revived session, using a model similar to the tabs commonplace in today's web browsers. A revived session operates as a normal desktop session; its new execution can diverge from the sequence of events that occurred in the original recording. The ability to revive a past session is analogous to how a modern laptop can resume operation after a period of hibernation to disk. DejaView extends this concept by allowing simultaneous revival of multiple past sessions, that can run side-by-side independently of each other and of the current session. The user can copy and paste content amongst her active sessions.

Recording a user's computer activity raises valid privacy and security concerns [5], as this information could be exploited to infringe upon the user's civil liberties or for criminal purposes. To mitigate some of the security concerns, user input is not directly recorded; only the changes it effects on the display are kept. Standard encryption techniques can also be used to provide an additional layer of protection. Addressing the larger privacy and security ramifications of this computing model is beyond the scope of this paper.

3. ARCHITECTURE OVERVIEW

To support its personal virtual computer recorder usage model, DejaView needs to record both the display and execution of a user's desktop computing environment such that the record can be played and manipulated at a later time. DejaView must provide this functionality in a manner that is transparent, has minimal impact on interactive performance, can preserve visual display fidelity, and is space efficient. DejaView achieves this by using a virtualization architecture that consists of two main components: a virtual display based on THINC [2] and a virtual execution environment based on Zap [28, 21]. These components leverage existing system interfaces to provide transparent operation without modifying, recompiling, or relinking applications, window systems, or OS kernels.

DejaView's virtual display decouples the display state from the underlying hardware and enables the display output to be redirected anywhere, making it easy to manipulate and record. DejaView operates as a client-server architecture and transparently provides a virtual display by leveraging the standard video driver interface, a well-defined, low-level, device-dependent layer that exposes the video hardware to the display system. Instead of providing a real driver for a particular display hardware, DejaView introduces a virtual display driver that intercepts drawing commands, records them, and redirects them to the DejaView client for display. DejaView uses the THINC display command set and display driver, but with enhancements for recording and local operation. All persistent display state is maintained by the display server; clients are simple and stateless. By allowing display output to be redirected anywhere, this approach also enables the desktop to be accessed both locally and remotely, which can be done using a wide range of devices given the client's simplicity.

DejaView's virtual execution environment decouples the user's desktop computing environment from the underlying OS, enabling an entire live desktop session to be continuously checkpointed, and later revived from any checkpoint. Building on Zap, DejaView leverages the standard interface between applications and the OS to transparently encapsulate a user's desktop computing session in a private virtual namespace. This namespace is essential to support DejaView's ability to revive checkpointed sessions. By providing a virtual namespace, revived sessions can use the same OS resource names as used before being checkpointed, even if they are mapped to different underlying OS resources upon revival. By providing a private namespace, revived sessions from different points in time can run concurrently and use the same OS resource names inside their respective namespaces, yet not conflict among each other. This lightweight virtualization mechanism imposes low overhead as it operates above the OS instance to encapsulate only the user's desktop computing session, as opposed to an entire machine instance. By using a virtual display and running its virtual display server inside its virtual execution environment, DejaView ensures that all display state is encapsulated in the virtual execution environment so that it is correctly saved at each checkpoint. Furthermore, revived sessions can then operate concurrently without any conflict for display resources since each has its own independent display state.

Building upon its core virtualization architecture, DejaView provides recording tools to save the display and execution state of the desktop, and playback tools to manipulate and interact with this recorded state. Two sets of desktop state are recorded at all times. The first consists of all visual output generated by the desktop, which allows users to quickly browse and playback recorded content. The second consists of all the application and file system state of the desktop, which allows users to revive their desktop as it was at any point in the past. Revived sessions behave just like the main desktop session, and users are free to continue to interact with them and possibly diverge from the path taken in the original recording. Multiple sessions can coexist since sessions are completely isolated from each other.

4. DISPLAY

4.1 Display Recording

DejaView leverages our previous work on the THINC [2] virtual display architecture to display and record visual output simultaneously. In particular, generated visual output is duplicated into a stream for display by the viewer, and a stream for logging to persistent storage. Both streams use the same set of commands (specifically the THINC display protocol commands), enabling both efficient storage and quick playback. Since display records are just collections of display commands, the display record can be easily replayed either locally or over the network using a simple application similar to the normal viewer.

DejaView can easily adjust the recording quality in terms of both the resolution and frequency of display updates without affecting the output to the user. Using THINC's screen scaling ability, the display can be resized to accommodate a wide range of resolutions. For example, the display can be resized to fit the screen of a PDA even though the original resolution is that of a full desktop screen. The recorded commands are resized independently, so a user can have the recorder save display output at full screen resolution even if she is currently viewing it at a reduced resolution to accommodate a smaller access device. The user can then go back and view the display record at full resolution to see detailed information that may not have been visible when viewed on the smaller device. Similarly, the user can reduce the resolution of the display commands being recorded to reduce its storage requirements. The user can also limit the frequency at which updates are recorded by taking advantage of THINC's ability to queue and merge display commands so that only the result of the last update is logged.

DejaView records display output as an append-only log of THINC commands, where recorded commands specify a particular operation to be performed on the current contents of the screen. DejaView also periodically saves full screenshots of the display for the following two reasons. First, it needs a screenshot to provide the initial state of the display that subsequent recorded commands modify. Second, if a user wants to display a particular point in the timeline, DejaView can start with the closest prior screenshot and only replay a limited number of commands, thereby enabling desktop session browsing at real-time speeds. DejaView records display output in a manner similar to an MPEG movie where screenshots represent self-contained independent frames from which playback can start, and commands in the log represent dependent frames which encode a change relative to the current state of the display. Since screenshots consume significantly more space, and they are only required as a starting point for playback, DejaView only takes screenshots at long intervals (e.g. every 10 minutes) and only if the screen has changed enough since the previous one.

By using display protocol commands for recording, DejaView ensures that only those parts of the screen that change are recorded, thus ensuring that the amount of display state saved only scales with the amount of display activity. If the screen does not change, no display commands are generated and nothing is recorded. The virtual display driver knows not only which parts change, but also how they change. For example, if the desktop background is filled with a solid color, DejaView can efficiently represent this in the record as a simple `solid fill` command. In contrast, regularly taking snapshots of the full screen would waste significant processing and storage resources as even the smallest of changes, such as the clock moving to the next second, would trigger a new screenshot. It could be argued that the screenshots could be compressed on the fly using a standard video codec, which could convert a sequence of screenshots into a series of smaller differential changes. However, this additional computation significantly increases the overhead of the system and may not provide a desirable tradeoff between storage and display quality for the synthetic content of desktop screens. In contrast, DejaView's approach knows precisely what changes, what needs to be saved, and the best representation to use when saving it.

DejaView indexes recorded command and screenshot data using a special `timeline` file that is used to quickly locate the screenshot associated with a given time. This file consists of chronologically ordered, fixed-size entries of the time at which a screenshot was taken, the file location in which its data was stored, and the file location of the first display command that follows that screenshot. This organization allows for fast playback over the recorded data as described in Section 4.3.

4.2 Text Capture

In addition to visual output, DejaView records contextual information by capturing all text that is displayed on the screen and using it as an index to the display record. Contextual information includes data such as the on-screen text, the window that it came from, the duration in which the text was on the screen, etc. Because there is a wide array of application-specific mechanisms used for rendering text, capturing textual information from display commands is often not possible. We considered using optical character recognition (OCR) on display records, but found currently available OCR technology to be slow and inaccurate for typical desktop screen contents. Instead, DejaView leverages ubiquitous accessibility mechanisms provided by most modern desktop environments and widely used by screen readers to provide desktop access for visually-impaired users [14]. These mechanisms are typically incorporated into standard GUI toolkits, making it easy for applications to provide basic accessibility functionality. DejaView uses this infrastructure to obtain both the text displayed on the screen and useful context, including the name and type of the application that generated the text, window focus, and special properties about the text (e.g. if it is a menu item or an HTML link). By using a mechanism natively supported by applications, DejaView has maximum access to textual information without requiring any application or desktop environment modifications.

DejaView uses a daemon to collect the text on the desktop and index it in a database that is augmented with a text search engine. At the most basic level, the daemon behaves very similarly to a screen reader, as both programs have similar functional requirements. At startup time, the daemon registers with the desktop environment and asks it to deliver events when new text is displayed or existing text on the screen changes. As events are received, the daemon wakes up, collects the new text and state from the application, and inserts this information into the database.

Throughout the data collection process, DejaView's daemon needs to be mindful of any overhead it creates on the interactive performance of the desktop. In particular, two aspects of the accessibility mechanism need to be handled with care. First, events are delivered synchronously, meaning that applications block until event delivery is finished. Second, the accessible components of applications are stored as trees. These trees can grow as UI complexity increases, and are extremely expensive to traverse; only one component in the tree can be accessed at any point in time, and accessing each component requires continuous context switching between the daemon and the application.

DejaView's daemon minimizes both event processing time and the number of queries to applications by keeping a number of data structures that exactly mirror the accessible state of the desktop. At startup, the daemon traverses all the applications, and builds its own mirror tree. This tree is used to keep an exact replica of the state of the desktop, which can be traversed at a tiny fraction of the cost of traversing the real accessible tree; the latter can take a couple seconds and destroy interactive responsiveness. To minimize event processing time, a hash table maps accessible components to nodes in the mirror tree. As events are received, the daemon can quickly query the corresponding node and determine which parts of the tree need to be updated.

Keeping an exact replica of the text state of the desktop also yields data that is valuable in providing searching capabilities to the recorded content. As events are generated, the full contents of the tree are inserted and indexed into the database. By indexing the full state of the desktop's text over time, DejaView is able to access the temporal relationships and state transitions of all displayed text as database queries. Consider, for example, a user that is looking for the time when she started reading a paper, but all she recalls is that a particular web page was open at the same time. If text was only indexed when it first appeared on the screen, this temporal relationship between the web page and the paper would never have been recorded and the user would be unable to access the content of interest. DejaView's indexing strategy also allows it to infer text persistence information that can be used as a valuable ranking tool. For example, a user could be less interested in those parts of the record when certain text was always visible, and more interested in the records where the text appeared only briefly.

A limitation of our approach is that not every application may provide an accessibility interface. For example, while DejaView can capture text information from PDF documents that are opened using the current version of Adobe Acrobat Reader, other PDF viewers used in Linux do not yet provide an accessibility interface. However, our experience has been that most applications do not suffer from this problem, and there is an enormous impetus to get accessibility interfaces into all desktop applications to provide universal access. The needs of visually impaired users will continue to be a driving force in ensuring that applications increasingly provide accessibility interfaces, enabling DejaView to extract textual information from them.

4.3 Playback

Visual playback and search are performed by the DejaView client. Various time-shifting operations are supported, such as skipping to a particular time in the display record, and fast forward or rewind from one point to another. To skip to any time T in the record, DejaView uses fast binary search over the `timeline` index file to look for the entry with the maximum time less than or equal to T. Once the desired entry is found, DejaView uses the entry's screenshot information to access the screenshot data and use it as the starting point for playback. Subsequently, it uses the entry's command information to locate the command that immediately follows the recovered screenshot. Starting with that command, DejaView processes the list of commands up to the first command with time greater than T. DejaView builds a list of commands that are pertinent to the contents of the screen by discarding those that are overwritten by newer ones, thus minimizing the time spent in the playback operation. The list is ordered chronologically to guarantee correct display output. After the list has been pared of the irrelevant commands, each command on the list is retrieved from the corresponding files and displayed.

To play the display record from the current display until time T, DejaView simply plays the commands in the command file until it reaches a command with time greater than T. DejaView keeps track of the time of each command and sleeps between commands as needed to provide playback at the same rate at which the session was originally recorded. DejaView can also playback faster or slower by scaling the time interval between display commands. For example, it can provide playback at twice the normal rate by only allowing half as much time as specified to elapse between commands. To playback at the fastest rate possible, DejaView ignores the command times and processes them as quickly as it can. Except for the accounting of time, the playback application functions similarly to the DejaView viewer in processing and displaying the output of commands.

To fast forward from the current display to time T, DejaView reads the `timeline` index file and plays each screenshot in turn until it reaches a screenshot with time greater than T. It then finds the tuple in the `timeline` file with the maximum time less than or equal to T, which corresponds to the last played screenshot, and uses the tuple to find the corresponding next display command in the command file. Starting with that command, DejaView plays all subsequent commands until it reaches a command with time greater than T. Rewind is done in a similar manner except going backwards in time through the screenshots.

4.4 Search

In addition to standard PVR-like functionality, DejaView provides a mechanism that allows users to quickly search recorded display output. DejaView search uses the index built from captured text and contextual information to find and return relevant results. In the simplest case, DejaView allows users to perform simple boolean keyword searches, which will locate the times in the display record in which the query is satisfied. More advanced queries can be performed by specifying extra contextual information. A useful query users have at their disposal is the ability to tie keywords to applications they have used or to the whole desktop. For example, a user may look for a particular set of words limited to just those times when they were displayed inside a Firefox window, and further narrow the search by adding the constraint that a different set of words be visible somewhere else on the desktop or on another application. Users can also limit their searches to specific ranges of time or to particular actions. For example, a user may search for results only on a given day and only for text in applications that had the window focus. A full study of how desktop contextual information can be used for search is beyond the scope of this paper.

Another search mechanism is provided through annotations. At the most basic level, annotations can be simply created by the user by typing text in some visible part of the screen since the indexing daemon will automatically add it to the record stream. However, the user may have to provide some unique text that will allow the annotation to stand out from the rest of the recorded text. To help users in this case, DejaView provides an additional mechanism which takes further advantage of the accessibility infrastructure. To explicitly create an annotation, the user can write the text, select it with the mouse, and press a combination key that will message the indexing daemon to associate the selected text with an attribute of *annotation*. The indexing daemon is able to provide this functionality transparently since both text selection and key strokes events can be delivered by the accessibility infrastructure.

Search results are presented to the user in the form of a series of text snippets and screenshots, ordered according to several user-defined criteria. These include chronological ordering, persistence information (ie. how long the text was on the screen), number of times the words appear, and so on. The search is conducted by first passing a query into the database that results in a series of timestamps where the query is satisfied. These timestamps are then used as indices into the display stream to generate screenshots of the user's desktop. The operation is very similar to the visual playback described in Section 4.3, with the difference being that it is done completely offscreen, which helps speed up the operation. DejaView also caches screenshots for search results, using a LRU scheme, where the cache size is tunable. This provides significant speedup in cases where the user has to continuously go back to specific points in time.

Each screenshot generated is a portal through which users can either quickly glance at the information they were looking for, or, by simply pressing a button, revive their desktop session as it was at that particular point in time. In addition, when the query is satisfied over a contiguous period of time, the result is displayed in the form of a first-last screenshot, which, borrowing a term from Lifestreams [10], represents a *substream* in the display record. Substreams behave like a typical recording, where all the PVR functionality is available, but restricted to that portion of time.

5. LIVE EXECUTION

5.1 Record

DejaView's virtual execution environment decouples the user's desktop computing session from the underlying OS instance so that it can be recorded by continuously checkpointing all the OS state associated with the session. DejaView's checkpoints are time stamped, enabling a user to select a point in time from the display record to revive the corresponding checkpoint.

DejaView checkpointing must satisfy two key requirements. First, it must provide a coordinated and consistent checkpoint of the execution environment and the many processes and threads that constitute a desktop environment and its applications; this is quite different from just checkpointing a single process. Second, it must have minimal impact on interactive desktop performance. To address these requirements, DejaView takes a globally consistent checkpoint across all processes in the user's desktop session while all processes are stopped so that nothing can change, but then minimizes the type and cost of operations that need to occur while everything is stopped.

5.1.1 Consistent User Desktop Checkpointing

DejaView runs a privileged process outside of the user's virtual execution environment to perform a consistent checkpoint of the session in four basic steps. First, the session is quiesced and all its processes are forced into a stopped state, to ensure that the saved state is globally consistent across all processes in the session. Second, the execution state of the virtual execution environment and all processes is saved. Third, a file system snapshot is taken to provide a version of the file system consistent with the checkpointed process state. Fourth, the session is resumed.

Using a separate process makes it easier to provide a globally consistent checkpoint across multiple processes in a user's session by simply quiescing all processes then taking the checkpoint; this avoids the complexity of having to synchronize the checkpoint execution of multiple processes, should they checkpoint themselves independently. Furthermore, if a process cannot run, for example if it is stopped waiting for the completion of the `vfork` system call, it cannot perform its own checkpoint. DejaView's design allows it to checkpoint at any time, even when not all processes are runnable. Further details on how DejaView checkpoints process state consistently across multiple processes are presented in previous work by two of the authors [21].

DejaView needs to capture a snapshot of the file system state at every checkpoint since the process execution state depends on the file system state. For example, if a process in a user's session is using the file `/tmp/foo` and is checkpointed at time T, it would be impossible to revive the user's session from time T if the file was later deleted and could not be restored to its state at time T. Furthermore, DejaView needs to be able to save the file system state quickly without interrupting the user's interaction with the system.

Approaches such as `rsync` [38], LVM [24], or logging file system related system calls could be considered for saving the file system state, but these have various performance or functionality limitations. DejaView takes a simpler and more efficient approach by leveraging file systems that provide native snapshot functionality, in which operations never overwrite the state of an existing snapshot. Specifically, DejaView uses a log structured file system [20], in which all file system modifications append data to the disk, be it meta data updates, directory changes or syncing data blocks. Thus, every modifying transaction results in a file system snapshot point. DejaView creates a unique association between file system snapshots and checkpoint images by storing a counter that is incremented on every checkpoint in both the checkpoint image's meta data and the file system's logs. To restore the file system, DejaView simply selects the snapshot identified by the counter found in the checkpoint image, and creates an independent writable view of that snapshot.

5.1.2 Optimizing for Interactive Performance

DejaView checkpoints interactive processes without impacting the user's perception of the system by minimizing downtime due to processes being stopped in two ways. First, it shifts expensive I/O operations outside of the window of time when processes are stopped so that they can be done without blocking user interactivity. Second, it employs various optimizations to minimize the cost of operations that do occur while processes are stopped.

DejaView employs three optimizations to shift the latency of expensive I/O operations before and after the window of time when processes are stopped. First, DejaView performs a file system synchronization before the session is quiesced. While file system activity can occur between this *pre-snapshot* and the actual file system snapshot, it greatly reduces, and many times eliminates, the amount of data needed to be written while the processes are unresponsive.

Second, before DejaView quiesces the session by sending all the processes a stop signal, DejaView attempts to ensure that the processes are able to handle the signal promptly, which we call *pre-quiescing.* If a process is blocked in an uninterruptible state, such as when performing disk I/O, it will not handle the signal until the blocking operation is complete. Meanwhile the rest of the processes will have already been stopped, which may be noticeable to the user. Therefore, DejaView waits to quiesce the session until either all the processes are ready to receive signals or a configurable time has elapsed.

Third, since disk throughput is limited, DejaView defers writing the persistent checkpoint image to disk until after the session has been resumed. Instead, the checkpoint is first held in memory buffers that DejaView preallocates. DejaView estimates the size of the buffer based on the average amount of buffer space actually used for recent checkpoints.

DejaView employs three optimizations to reduce downtime while processes are stopped. First, to reduce downtime due to copying memory blocks as well as the amount of memory required for the checkpoint, DejaView leverages copy-on-write (COW) techniques to enable it to defer the memory copy until after the session has resumed. Instead of creating an explicit copy of the memory while the session is quiesced, DejaView marks the memory pages as COW. Since each memory page is automatically copied when it is modified, DejaView is able to get a consistent checkpoint image, even after the session has been resumed.

Second, to avoid the overhead of saving the contents of unlinked files that are still in use, DejaView relinks such files within the same file system before the file system snapshot is performed. Since deleted files are removed from their parent directory, their contents are not readily accessible on revive for DejaView to open. However, their contents remain intact for as long as the files remain in use. Relinking ensures that these contents remain accessible without explicitly saving them to the checkpoint image. To avoid namespace conflicts, the files are relinked within a special directory that is not normally accessible within the virtual execution environment. When the session is revived, DejaView temporarily enables the files to be accessible within the user's session, opens the files and immediately unlinks them, restoring the state to what it was at the time of the checkpoint.

Third, since the memory state of the processes dominates the checkpoint image, DejaView provides an incremental checkpoint [8] mechanism that reduces the amount of memory saved by only storing the parts of memory that have been modified since the last checkpoint. This optimization reduces processing overhead since less pages need to be scanned and saved to memory, and reduces storage requirements since fewer pages need to be written to disk. For DejaView to operate transparently and efficiently, we leverage standard memory protection mechanisms available on all modern operating systems. The basic mechanism used by DejaView is to mark saved regions of memory as read-only and then intercept and process the signals generated when those regions are modified.

During a full checkpoint, all the process's writable memory regions are made read-only. DejaView marks these regions with a special flag to distinguish them from regular read-only regions. After the process is resumed, any attempts to modify such regions will cause a signal to be sent to the process. DejaView intercepts this signal and inspects the region's properties. If it is read-only and marked with the special flag, then DejaView removes the flag, makes the region writable again, and resumes the process without delivering the signal. If the flag is not present, the signal is allowed to proceed down the normal handling path. During the next incremental checkpoint, only the subset of memory regions that have been modified is saved. DejaView is careful to handle exceptions that occur when writing a marked region during system call execution to ensure that the system call does not fail in this case. This case cannot be handled by user-level checkpointing techniques [23, 33, 31] since, instead of a signal being passed to the process, an error is returned to the caller of the system call function. However, using user-level interposition to monitor such system calls is non-atomic and thus subject to race conditions [11].

DejaView's incremental checkpoint implementation does not restrict applications from independently invoking system calls that affect the memory protection and the memory layout of a process (e.g. `mprotect`, `mmap`, `munmap`, `mremap`). DejaView intercepts those calls to account for the changes they impose on the layout. For example, if the application unmaps or remaps a region, that region is removed or adjusted in the incremental state. Likewise, if it changes the protection of a region from read-write to read-only then that region is unmarked to ensure that future exceptions will be propagated to the application.

Checkpoints are incremental by default to conserve space and reduce overhead, but full checkpoints are taken periodically when the system is otherwise idle. This is for redundancy and to reduce the number of incremental checkpoint files needed to revive a session. For example, if full checkpoints are on average ten times larger than incremental checkpoints, a full checkpoint every thousand incremental ones only incurs an additional 1% storage overhead.

5.1.3 Checkpoint Policy

DejaView needs to record enough state to enable a user to revive any session that can be accessed from the display record, while maintaining low overhead. Given the bursty nature of desktops, where user input may trigger a barrage of changes followed by long idle times, the naive approach of taking checkpoints at regular intervals is suboptimal. It would miss important updates that occurred in the interval, while wastefully recording during periods of inactivity.

Instead, DejaView checkpoints in response to actual display updates. Since checkpointing is more expensive than recording the display, DejaView minimizes overhead in two ways. First DejaView reduces runtime overhead by limiting the checkpoint rate to at most once per second by default. The rate can be limited because display activity consists of many individual display updates, but the user only notices their aggregate effect.

Second, to reduce storage requirements, DejaView uses a default checkpoint policy that employs three optimizations. First, it disables checkpointing in the absence of user input when certain applications are active in full screen mode. For instance, DejaView skips checkpoints when the screensaver is active or when video is played in full screen mode since checkpoints are either unlikely to be of interest or do not add any useful information beyond the display record.

Second, even if the display is modified, DejaView skips checkpoints if display activity remains below a user defined threshold, for example, if only a small portion of the display is affected (by default, at most 5% of the screen). This is useful to disregard trivial display updates that are not of much interest to the user, such as the blinking cursor, mouse movements, or clock updates.

Third, even when display activity may be low, DejaView still enables checkpoints in the presence of keyboard input (e.g. text editing), to allow users to return to points in time at which they generated their data. In this case the policy reduces the rate to once every ten seconds to match the expected amount of data generated by the user, which is limited by her typing speed. For an average person who types 40 words per minute [29], this checkpoint rate translates to a checkpoint roughly every 7 words. This is more than sufficient to capture most document word processing of interest.

Note that the checkpoint policy is flexible in that the user may tune any of the parameters. The policy is also extensible and can include additional rules. For example, a user may add a control that would disable checkpoints when the load of the computer rises above a certain level.

5.2 Revive

DejaView allows a user to browse and search the display record and revive the desktop session that corresponds to that point in time. To revive a specific point in time, DejaView searches for the last checkpoint that occurred before that point in time. Since the desktop session is revived at a slightly earlier time than the selected display record, it is possible that some differences exist in the live display of the revived desktop and the static display record. While one could log all events during execution to support deterministic replay of the desktop, DejaView does not do this because of the extra complexity and overhead. More importantly, such visual differences are not noticeable by the user since checkpoints rate can reach once per second if necessary. When the session is revived, if the display was changing slowly, there will be minimal visual differences. On the other hand, if the display was changing rapidly, the display will continue to change, making any initial differences inconsequential.

Reviving a checkpointed desktop session's consists of the following steps. First, a new virtual execution environment is created. Second, the file system state is restored as described below. Third, a forest of processes is created to match the set of processes in the user's session when it was checkpointed, and the processes then execute to restore their state from the checkpoint image. This state includes process run state, program name, scheduling parameters, credentials, pending and blocked signals, CPU registers, FPU state, `ptrace` information, file system namespace, list of open files, signal handling information, and virtual memory. Once all processes have been restored, they are resumed and allowed to continue execution. DejaView then signals the viewer application to create a new connection to the revived session, which is displayed in a new window in the viewer.

For the incremental checkpoints, reviving the user's session requires accessing a set of incremental checkpoint files instead of a single checkpoint image file. To revive the session, DejaView starts by reading in data from the current (time selected) checkpoint image. When the restoration process encounters a memory region that is contained in another file, as marked by its list of saved memory regions, it opens the appropriate file and retrieves the necessary pages to fill in that portion of memory. This process then continues reading from the current checkpoint image, reiterating this sequence as necessary, until the complete state of the desktop session has been reinstated.

Standard snapshotting file systems only provide read-only snapshots, which may be useful for backup purposes, but are ill-suited for supporting a revived session that requires read-write semantics for its normal operation. To provide a read-write file system view, DejaView leverages unioning file systems [46] to join the read-only snapshot with a writable file system by stacking the latter on top of the former. This creates a unioned view of the two: files system objects, namely files and directories, from the writable layer are always visible, while objects from the read-only layer are only visible if no corresponding object exists in the other layer.

While operations on objects from the writable layer are handled directly, operations on objects from the read-only layer are handled according to the their specific nature. If the operation does not modify the object, it is passed to the read-only layer. Otherwise, DejaView first creates a copy of the object in the writable layer, then handles the operation there. While copying an entire file can degrade file system performance when done often with large files, desktop applications typically do not modify large files; more commonly, they overwrite files completely, which obviates the need to copy the file between the layers.

DejaView's combination of unioning and file system snapshots provides a branchable file system to enable DejaView to create multiple revived sessions from a single checkpoint. Since each revived session is encapsulated in its own virtual execution environment and has its own writable file system layer, multiple revived sessions can execute concurrently. This enables the user to start with the same information, but to process it in separate revived sessions in different directions. Furthermore, by using the same log structured file system for the writable layer, the revived session retains DejaView's ability to continuously checkpoint session state and later revive it.

Analogous to resuming a hibernated laptop, the user does not expect external network connections to remain valid after DejaView revives a session since the state of the peers can not be guaranteed. Thus, when reviving a session, DejaView drops all external connections of stateful protocols, such as TCP, by resetting the state of their respective sockets; internal connections that are fully contained within the user's session, e.g. to *localhost*, remain intact. For the application, this appears as a dropped network connection or a disconnect initiated by the peer, both of which are scenarios that applications can handle gracefully. For instance, a web browser that had an open TCP connection to some web server would detect that the connection was dropped and attempt to initiate a new connection. The browser will be able to load new pages as the user clicks on hyperlinks in a manner that is transparent to the user. Similarly, a revived secure shell (ssh) will detect the loss of connection, and report an error to the user. On the other hand, sockets that correspond to stateless protocols, such as UDP, are always restored precisely since the underlying operating system does not maintain any protocol specific state that makes assumptions about, or requires the cooperation of a remote peer.

By default, network access is initially disabled in a revived session to prevent applications from automatically reconnecting to the network and unexpectedly losing data as a result of synchronizing their state with outside servers. For example, a user who revived a desktop session to read or respond to an old email that had been deleted on the outside mail server would not want her email client to synchronize with that server and lose the old email. However, the user can re-enable network access at any time, either for the entire session, or on a per application basis. Alternatively, the user can configure a policy that describes the desired network access behavior per application, or select a preset one. For new applications that the user launches within the revived session, network access is enabled by default.

6. EXPERIMENTAL RESULTS

We have implemented a DejaView prototype system for Linux desktop environments. The server prototype consists of a virtual display driver for the X window system that provides display recording, a set of user-space utilities and loadable kernel modules for off-the-shelf Linux 2.6 kernels that provide the virtual execution environment and the ability to checkpoint and revive user sessions, and a snapshotable and branchable file system based on NILFS [20] and UnionFS [46] that guarantees consistency between checkpoints and file system state. For capturing text informa-

Name	Description
web	Firefox 2.0.0.1 running iBench web browsing benchmark to download 54 web pages
video	MPlayer 1.0rc1-4.1.2 playing *Life of David Gale* MPEG2 movie trailer at full-screen resolution
untar	Verbose untar of 2.6.16.3 Linux kernel source tree
gzip	Compress a 1.8 GB Apache access log file
make	Build the 2.6.16.3 Linux kernel
octave	Octave 2.1.73 (MATLAB 4 clone) running Octave 2 numerical benchmark
cat	`cat` a 17 MB system log file
desktop	16 hr of desktop usage by multiple users, including Firefox 2.0.0.1, GAIM 1.5, OpenOffice 2.0.1, Adobe Acrobat Reader 7.0, etc.

Table 1: Applications scenarios

tion, DejaView uses the accessibility infrastructure of the GNOME desktop environment [14]. Indexing and searching text is performed using the Tsearch extension [39] for the PostgreSQL database system. A simple client viewer is used to access the DejaView desktop locally or remotely and provides display browse and search functions.

Using this prototype, we present experimental data that quantifies the performance of DejaView when running a variety of common desktop applications. We present results for both application benchmarks and real user desktop usage. We focus on quantifying the storage requirements and performance overhead of using DejaView in terms of the cost of continuously recording display and execution. For the application benchmark experiments, we did full fidelity display recording and checkpoint once per second to provide a conservative measure of performance. For the real user desktop usage experiments, we did full fidelity display recording and checkpoint according to the policy described in Section 5.1.3 to provide a corresponding real world measure of performance. We also measured the overhead of our virtual display mechanism and virtual execution environment and found it to be quite small; we omit these results due to space constraints.

We used the desktop application scenarios listed in Table 1. We considered several individual application scenarios running in a full desktop environment, including scenarios that created lots of display data (`web`, `video`, `untar`, `make`, `cat`) as well as those that did not and were more compute intensive (`gzip`, `octave`). These scenarios measure performance only during periods of busy application activity, providing a conservative measure of DejaView performance since real interactive desktop usage typically consists of many periods during which the computer is not utilized fully. For example, our `web` scenario downloads a series of web pages in rapid fire succession instead of having delays between web page downloads for user think time to stress DejaView and measure its worst-case performance. To provide a more representative measure of performance, we measured real user desktop usage (labeled as `desktop` in the graphs) by aggregating data from multiple graduate students using our prototype for all their computer work over many hours.

For all our experiments the DejaView viewer and server ran on a Dell Dimension 5150C with a 3.20 GHz Intel Pentium D CPU, 4 GB RAM, a 500 GB SATA hard drive and connected to a public switched Fast Ethernet network. The machine ran the Debian Linux distribution with kernel version 2.6.11.12 using X.org 7.1 as the window system, and GNOME 2.14 as the desktop environment. The display resolution was 1024x768 for the application benchmarks and 1280x1024 for real desktop usage measurements. For our web application scenario, we also used an IBM Netfinity 4500R server with dual 933 MHz Pentium III CPUs and 512 MB RAM as the web server, running Linux kernel version 2.6.10 and Apache 1.3.34.

Figure 2 shows the performance overhead of running DejaView for each application scenario. We ran each scenario without recording, with each of the individual recording components only, and with full recording, including display, text indexing, and checkpoints once per second. Performance is shown normalized to the execution time without any recording. The results show that there is some overhead for recording, but in practice there were no visible interruptions in the interactive desktop experience and real-time interaction was not affected. Full recording overhead is small in almost all scenarios, including those that are quite display intensive such as `cat` and full-screen video playback. In all cases other than web browsing, the overhead was less than 20%. For `video`, the most time-critical application scenario, the overhead of full recording is less than 1% and does not cause any of the video frames to be dropped during display. For web browsing, the overhead was about 115% because the average download latency per web page was slightly more than half a second with full recording while it was .28 seconds without recording. We discuss the reasons for this overhead below. However, real users do not download web pages in rapid fire succession as the benchmark does, and the page download latencies with full recording are well below the typical one second threshold needed for users to have an uninterrupted browsing experience [25]. The web performance of DejaView with full recording is fast enough in practice for interactive web browsing. We did not measure the performance overhead of the `desktop` scenario given the lack of precise repeatability with real usage.

Figure 2 shows also how the DejaView recording components individually affect performance. Both display and checkpoint recording overhead are small in all scenarios, including those that are quite display intensive such as `cat` and full-screen video playback. The largest display recording overhead is 9% for the rapid fire web page download, which changes almost all of the screen continuously and causes the web browser and DejaView server and viewer to compete for CPU and I/O resources. The display overhead for all other cases is less than 2%. As expected, `gzip` and `octave` have essentially zero display recording overhead since they produce little visual output. Interestingly, `video` has one of the smallest display recording overheads of essentially zero. Even though it changes the entire display for each video frame, it requires only one command for each video frame, resulting in 24 commands per second, a relatively modest rate of processing. For checkpoint recording, the largest overhead is for `make`, which is 13%. For other applications, the checkpoint overhead is less than 5%. In practice, the overhead is not typically noticeable to the user. Note that these checkpoint overheads were for once per second checkpointing and represent a conservative measure; the use of the checkpoint policy in practice would reduce checkpoint overhead even further.

Figure 2 additionally shows the index recording overhead, which is small in all scenarios except for the `web` benchmark. The overhead is less than 4% for all cases except

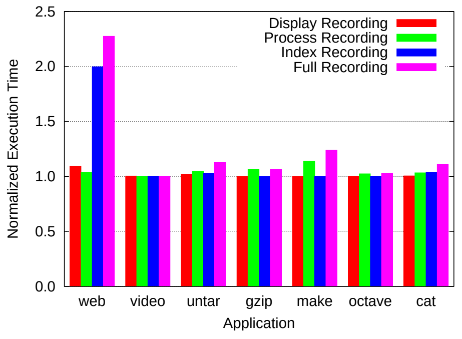

Figure 2: Recording runtime overhead

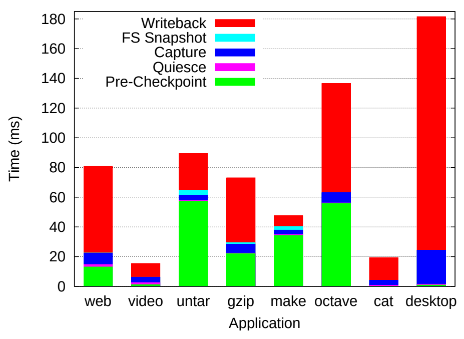

Figure 3: Total checkpoint latency

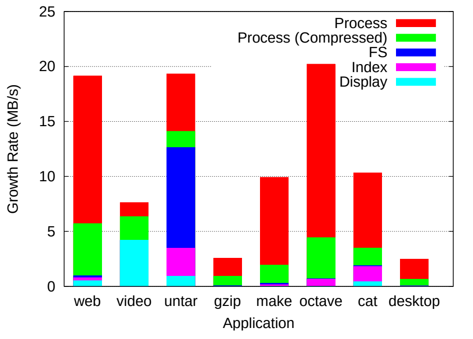

Figure 4: Recording storage growth

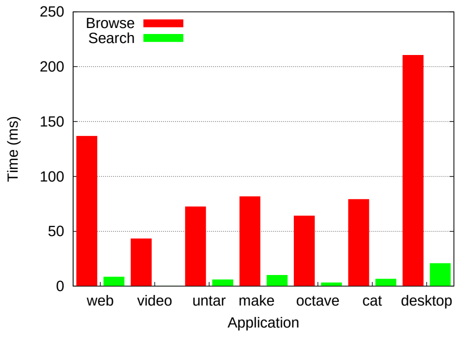

Figure 5: Browse and search latency

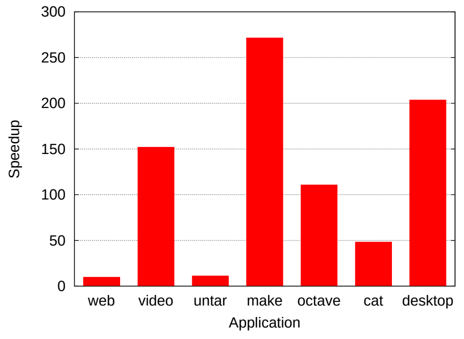

Figure 6: Playback speedup

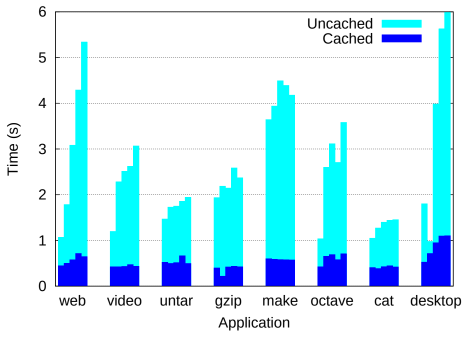

Figure 7: Revive latency

for the `web` benchmark. For the `web` benchmark, the indexing overhead is 99%, which accounts for almost all of the overhead of full recording. Unlike the other applications, the Firefox web browser creates its accessibility information on demand, rather than simply updating existing information. This dynamic generation of accessibility information coupled with a weakness in the current Firefox accessibility implementation results in much higher overhead when DejaView indexing records text information. We expect that this overhead will decrease over time as its accessibility features improve [9].

Figure 3 shows the average checkpoint times for each of the application scenarios. The times are broken down into five parts: *pre-checkpoint*, which includes pre-snapshot and pre-quiesce time, *quiesce*, *capture*, which is the time it takes to perform a copy-on-write capture of all memory and state, *file system snapshot*, and *writeback*, which is the time to write the data out to disk. Downtime due to checkpointing is the sum of quiesce, capture, and file system snapshot times. Overall, the results show that application downtime due to checkpoints is small enough that DejaView can perform full recording of live execution state without a noticeable degradation in interactive application performance.

Figure 3 shows that application downtime for DejaView checkpointing is minimal, less than 10 ms for all application benchmarks and roughly 20 ms on average for real desktop usage. Average downtime is higher for the real usage cases because the users often ran multiple applications at once, and the DejaView checkpoint policy results in fewer checkpoints, so each checkpoint can take longer due to an increased amount of changed state. Though an application is unresponsive while it is stopped, these results show that even the largest application downtimes are less than the typical system response time thresholds of 150 ms needed for supporting most human computer interaction tasks without noticeable delay [35]. For instance, for `video` the application downtime was only 5 ms, which is small enough to avoid interfering with the time-critical display of video frames.

Application downtime is primarily due to the copy-on-write capture of memory state, though file system snapshot time can account for up to half of the downtime as in the case of `untar`, which is more file system intensive. The downtime is minimized due to the incremental and COW checkpointing mechanisms, the pre-checkpoint operations, and deferring the writing of the checkpoint image to disk after the session has been resumed. For comparison purposes, we attempted the same experiments without these optimizations for minimizing downtime, but could not run them. The unoptimized mechanism was too slow to checkpoint at the once a second rate DejaView uses; it took too long to even write the checkpoint data to disk.

Figure 3 shows that pre-checkpoint and writeback account for most of the total average checkpoint time, which is under 100 ms in most cases but is as high as 180 ms for the more complex user desktop. Since pre-checkpoint and writeback overlap with application execution, they do not result in downtime that would interfere with interactive performance. The large majority of pre-checkpoint time is consumed by the file system pre-snapshot. Pre-quiesce is on average very small, but is essential because it has high variability and can be as large as 100 ms.

Figure 4 shows the storage space growth rate DejaView experiences for each of the application scenarios. We decompose the storage requirements into the amount of increased storage DejaView imposes for display state, display indexing, process checkpoint and file system snapshot state. For display, indexing, and process checkpoint state, we measure the size of the files created to store them. However, for file system snapshot state we report the difference between the entire snapshot file system usage and what is visible to the user at the end of the scenario, as the visible size is independent of DejaView. We approximate the visible size by creating an uncompressed tar archive of the visible state, resulting in a somewhat overestimate of the file system storage growth rate. Since process checkpoint state is easily compressible, we show both the storage growth rate for uncompressed checkpoints and compressed checkpoints by overlaying the latter on the former in the figure.

For all of the application scenarios except `video` and `untar`, DejaView storage usage is dominated by checkpoint sizes. Figure 4 shows that the storage growth rate for the scenarios range from 2.5 MB/s for `gzip` to 20 MB/s for `octave`, assuming uncompressed checkpoints. Using `gzip` to compress the checkpoints, the storage growth rate for `octave` drops to just over 4 MB/s. With compressed checkpoints, the storage growth rate of all the applications except `video` and `untar` drops below 6 MB/s. For `video`, display recording accounts for most of the storage growth at 4 MB/s. `Video` requires more extensive display storage since each event changes the entire display, even though it does not create a high rate of display events. `Video` also has a relatively high percentage of display state versus checkpoint state because it is primarily a single process application that does not create much new process state during its execution. For `untar`, file system storage accounts for most of the storage growth at 9 MB/s. It requires more extensive file system storage due to the extraction of a tar archive containing the Linux kernel source tree, which contains lots of small files. Since DejaView's log structured file system needs to be able to recreate any point in the checkpoint history, it includes more overhead for file creation. This can be viewed in opposition to `gzip` where, despite having its large file continually snapshotted, the file system usage is small.

More importantly, typical usage does not have as high of a growth rate, resulting in much lower storage requirements in practice. As shown in Figure 4, the storage space growth rate for real user desktop usage is much more modest at only 2.5MB/s with uncompressed checkpoints and 0.6 MB/s with compressed checkpoints. In comparison, HDTV PVRs require roughly 9 GB of storage per hour of recording, or 2.5 MB/s. While DejaView storage requirements are greater than HDTV PVRs during periods of intense application activity, the desktop scenario results indicate that in practice they are comparable to HDTV PVRs. Also, as disk storage densities continue to double each year and multi-terabyte drives become commonplace in PCs [30], the storage requirements of DejaView will become increasingly practical for many users.

The storage space growth rate of DejaView is low primarily because of the checkpoint policy. To quantify its effectiveness, we examined the checkpoint logs recorded during the desktop usage. We found that DejaView skipped the majority of the checkpoints, taking checkpoints on average only 20% of the time. In the remaining time the policy deferred checkpointing for 13% of the time due to lack of display activity, 69% due to low display activity, and 18% due to reduced checkpoint rate during period of text editing. We estimate that with no policy, the storage growth rate would exceed 3 MB/s for the compressed case. If we also account for idle time (during which the screensaver is running and DejaView skips checkpoints) the storage rate would exceed 6 MB/s.

We also conducted experiments that show DejaView's effectiveness at providing access to recorded content, by measuring its search, browse, and revive performance. We measured DejaView search performance by first indexing all displayed text for our application tests and desktop usage, each in its own respective database, then issuing various queries. For each application benchmark, we report the average query time for five single-word queries of text selected randomly from the respective database. For real desktop usage, we report the average query time for ten multi-word queries, with a subset limited to specific applications and time ranges, to mimic the expected behavior of a DejaView user. Figure 5 shows that on average, DejaView is able to return search results in no more than 10 ms for the application benchmarks and in roughly 20 ms for real desktop usage. These results demonstrate that the query times are fast enough to support interactive search. Another important measure of search performance is the relevance of the query results, which we expect to measure based on a user study; this is beyond the scope of this paper.

We measured browsing performance by using the display content recorded during our application benchmarks and accessing it at regular intervals. However, we were careful not to skew results in DejaView's favor, by eliminating points in the recording where less than 100 display commands were issued from the previous point. Eliminating these points makes sense since they belong to periods in which the system was not actively used, and hence are unlikely to be of interest to the user. Figure 5 shows that on average, DejaView can access, generate, and display the contents of the stream at interactive rates, ranging from 40 ms browsing times for `video` to 130 ms for `web`. For real desktop usage, browsing times are roughly 200 ms. These results demonstrate that DejaView provides fast access to any point in the recorded display stream, allowing users to efficiently browse their content.

To demonstrate how quickly a user can visually search the record, we measured playback performance of all the application scenarios and measured how long it would take to play the entire visual record. Figure 6 demonstrates that DejaView is able to playback an entire record at many times the rate at which it was originally generated. For instance, Figure 6 shows that DejaView is able to playback regular user desktops at over 200 times the speed it was recorded. While some benchmarks, in particular ibench, do not show as much of a speedup, we attribute this to the fact that they are constantly changing data at the rate of display updates. Even in the worst case, DejaView is able to display the visual record at over 10 times the speed at which it was recorded. These results demonstrate that DejaView can browse through display records at interactive rates.

For each of the application scenarios, Figure 7 shows the time it takes to revive the user's desktop session from when a user clicks on "Take Me Back" to when the desktop session is ready for use. Results are shown for using checkpoint files that are not cached as well as for cached. For the uncached case, revive times are all several seconds and are dominated by I/O latencies. For the cached case, revive times are all well under a second and commonly around half a second. These times provide a more direct measure of the actual processing time required to revive a session. Reviving using checkpoint files that have been cached due to recent file access more commonly occurs when users revive a session at a time relatively close to the current time.

We show the time to revive the user's session from five different points in time evenly spaced throughout the application's execution. For each application, the bars in the graph are ordered chronologically from left to right. The revive times from uncached checkpoint data show an increase over time, while those from cached checkpoint data are relatively constant across each application benchmark. Since incremental checkpointing is used, the revive times from checkpoints later in the application executions involve accessing more checkpoint files. However, the cost of accessing multiple files is not the reason for the increase in revive times here; reviving from non-incremental checkpoints would show a similar increase. The increase is instead largely due to increased memory usage by the applications as they execute. For reviving from uncached checkpoint files, the first revive time is often significantly faster than the others because the applications are not yet fully loaded. Subsequent uncached revive times reflect moderate growth for most applications because memory usage tends to increase over time, resulting in more saved memory state that needs to be read in from disk to revive the session. The `web` benchmark shows a substantial increase in revive times, growing by more than a factor of two from the second to the last revive. The reason for this is that the Firefox web browser is an application whose memory usage grows more dramatically during the benchmark, by more than a factor of two over its entire course. The uncached performance could be improved by demand paging; the current revive implementation requires reading in all necessary checkpoint data into memory before reviving. Reviving near the end of the application's execution is sometimes faster (e.g. `untar`) because the application is doing more work in the middle of its execution and using more memory than near the end. Overall, our results show that the cost of accessing multiple incremental checkpoint files while reviving a session is not prohibitive, and is outweighed by its ability to reduce more frequent and performance critical checkpoint times.

7. RELATED WORK

DejaView is created in the spirit of Vannevar Bush's Memex [4] vision to build a device that could store all of a user's documents and general information so that they could be quickly referenced. Inspired by the Memex vision, MyLifeBits [12] is centered around digitally capturing a lifetime of Gordon Bell's information with a focus on indexing and annotating individual documents. Lifestreams [10] was designed to minimize the time a user spends managing data by creating a time-ordered stream of documents in one's life as a replacement of the current desktop metaphor. All of these projects are complementary to DejaView. Neither approach provides DejaView's ability to record and index a user's computing experience such that it can be revived to consult the information it contains using its original native applications.

Desktop search tools, such as those from Google [15], Microsoft [44], and Yahoo [48], enable a user to search one's desktop files. Connections [36] improves a user's ability to search by extracting information that links what files were used with what programs and at what times to enable a user to perform a more contextualized search. The closest of these systems to DejaView is Stuff I've Seen (SIS) [6], which also uses information about what a user has seen in the context of search. While DejaView leverages the accessibility framework already built into GUI toolkits and applications, SIS requires writing gatherers and filters to extract contextual information for each application and data type that is used. Other projects such as Microsoft's WinFS [45] are attempts at replacing the traditional file system with a specialized document store that can be searched using natural language search mechanisms. All of these approaches focus only on searching files and typically only work for files in certain formats. They are largely orthogonal to DejaView.

Apple's Time Machine [1] enables a user to peruse and recover previous states in the file system. While Apple's Time Machine focuses solely on storage state, DejaView's wider scope includes display recording and playback, and allowing the user to search for state that has been seen but not committed to disk. Moreover, DejaView enables a user to revive and interact with a complete desktop session, not just manipulate old file data.

Screencasting provides a recording of a desktop's screen that can be played back at a later time [41, 47]. There have also been VNC-based approaches to recording desktop sessions [22], but most of them are tailored towards improving remote group collaboration. Screencasting works by screen-scraping and taking screenshots of the display many times a second. It requires higher overhead and more storage and bandwidth than DejaView's display recording, and the common approach of also using lossy JPEG or MPEG encoding to compensate further increases recording overhead, and decreases display quality. DejaView goes beyond screencasting by not only recording the desktop state, but by also extracting contextual information from it to enable display search. More recently, OpenMemex [26] is another system being independently developed that extends VNC-based screencasting to provide display search by using offline OCR to extract text from the recorded data. DejaView's use of available accessibility tools provides further contextual information, such as the application that generated the text, which is not available through OCR. Furthermore, DejaView provides the ability to revive and interact with a session instead of only viewing the display.

Virtual machines (VMs) have been used to log or snapshot entire OS instances and their applications to roll back execution to some earlier time [3, 7, 42, 43, 18]. Some of these approaches simply recreate the execution state at a previous point in time. Others deterministically replay execution exactly in uniprocessor environments and simpler VM models to enable debugging or intrusion analysis. However, implementing deterministic replay in practice without incurring prohibitive overhead remains a difficult problem especially for multiprocessor systems [40]. Moreover, this form of replay requires re-executing everything for playback, which may be desirable for debugging purposes but is expensive for PVR-functionality for information retrieval. In contrast, DejaView's display recording and playback provides deterministic replay of only display events for fast browsing and playback, while its revive functionality allows recreation of execution state at specific points in time. Unlike VM approaches, DejaView does not incur the higher overhead of logging or checkpointing entire machine instances, which is crucial to avoid degrading interactive desktop performance.

OS virtualization approaches like Capsules [34], Zap [28, 21] and OpenVZ [27] decouple processes from the underlying OS instance so that applications can be transparently checkpointed and restarted from secondary storage. DejaView differs from these approaches in reducing application downtime from checkpointing by up to two orders of magnitude, making continuous checkpointing of interactive desktops possible without degrading interactive performance. DejaView also differs in enabling desktops to be revived from any previous checkpoint, not just the most recent one. These benefits are achieved by correctly and completely supporting incremental and COW checkpoints of multithreaded and multiprocess cooperating desktop applications, moving potentially expensive file system, quiescing, and writeback operations out of the critical path to minimize application downtime, and providing file system snapshotting consistent with checkpointing to enable isolated execution from multiple checkpoints. Previous incremental process checkpointing approaches [32, 13, 17, 19] rely on the hardware's page protection mechanism exported by the OS to determine when a page has been written. DejaView uses the same approach but handles OS interactions not covered by other approaches. Previous process migration mechanisms introduced pre-copying [37] and lazy-copying [49] for reducing downtime due to copying virtual memory. DejaView builds on these ideas but applies them to checkpoint-restart and introduces them for reducing downtime due to saving non-memory OS resources and quiescing OS resources for checkpoint consistency.

8. CONCLUSIONS AND FUTURE WORK

DejaView introduces a new personal virtual computer recorder model to the desktop that enables What You Search Is What You've Seen (WYSIWYS) functionality to help users find, access, and manipulate information they have previously seen. The unique blend of functionality for display recording, playback, browsing, search, and reviving live desktop execution from any point in time are all transparently provided without any modifications to applications, window systems, or OS kernels. The key innovations that make this possible are (1) a virtual display mechanism to record and playback user-viewable interaction, (2) a text capture and indexing mechanism that makes novel use of accessibility interfaces so that recorded visual output can be searched, (3) checkpoint optimizations for recording application execution state without affecting interactive desktop performance, and (4) a coordinated checkpoint and file system mechanism that combines log structured and unioning file systems in a unique way to enable fast file system snapshots consistent with checkpoints, allowing checkpoints to be later revived for simultaneous read-write usage.

We have implemented a DejaView prototype and evaluated its performance on common desktop application workloads and with real desktop usage. Our results demonstrate that DejaView recording adds negligible overhead, capturing the display and execution state of interactive applications with only a few milliseconds of interruption, which is typically not noticeable to end users even for more time-sensitive applications such as movie playback. We show that DejaView's playback can enable users to quickly view display records at up to 270 times faster than real-time, and that browsing and searching display information is fast enough to be done at interactive rates. These results demonstrate that DejaView's personal virtual computer recorder provides WYSIWYS functionality fast enough for interactive use and without user noticeable performance degradation.

DejaView provides a new approach for information storage and retrieval and opens up new directions for future research. Some areas of future work include (1) conducting user studies to explore usage patterns to better understand how DejaView will be exploited by users over extended periods of time and how the user interface can be enhanced to better fit daily usage needs, (2) quantifying and improving the relevance and presentation of search results by exploring the use of desktop contextual information such as time, persistence, or the relationships among desktop objects, and (3) addressing the privacy and security ramifications of this emerging computing model.

9. ACKNOWLEDGEMENTS

Peter Chen and Alex Sherman provided helpful comments on earlier drafts of this paper. This work was supported in part by NSF ITR grant CNS-0426623, a DOE Early Career Award, AMD, and an IBM SUR Award. Oren Laadan was supported by a SOSP 2007 student scholarship from Infosys.

10. REFERENCES

[1] Apple - Mac OS X Leopard - Features - Time Machine. `http://www.apple.com/macosx/leopard/features/timemachine.html`.

[2] R. A. Baratto, L. N. Kim, and J. Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[3] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995.

[4] V. Bush. As We May Think. *The Atlantic Monthly*, July 1945.

[5] S. Carlson. On The Record, All the Time: Researchers digitally capture the daily flow of life. Should they? *The Chronicle of Higher Education*, 53(23):A30, Feb. 2007.

[6] S. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. C. Robbins. Stuff I've Seen: A System for Personal Information Retrieval and Re-use. In *Proceedings of the 26th Annual International ACM SIGIR Conference*, July - Aug. 2003.

[7] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[8] S. I. Feldman and C. B. Brown. IGOR: A system for Program Debugging via Reversible Execution. In *Proceedings of the 1988 ACM Workshop on Parallel and Distributed Debugging*, May 1988.

[9] Firefox Bug 372201. `https://bugzilla.mozilla.org/show_bug.cgi?id=372201`.

[10] E. T. Freeman. *The Lifestreams Software Architecture.* PhD thesis, Yale University, May 1997.

[11] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, Feb. 2003.

[12] J. Gemmell, G. Bell, and R. Lueder. MyLifeBits: A Personal Database for Everything. *Communications of the ACM*, 49(1):88–95, Jan 2006.

[13] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Nov. 2005.

[14] The GNOME Accessibility Project. `http://developer.gnome.org/projects/gap/`.

[15] Google Desktop Search. `http://desktop.google.com`.

[16] L. Gravano, P. Ipeirotis, and M. Sahami. QProber: A System for Automatic Classification of Hidden-Web Databases. *ACM Transactions on Information Systems*, 21(1), Jan. 2003.

[17] A. Joshi, W. Bridge, J. Loaiza, and T. Lahiri. Checkpointing in Oracle. In *Proceedings of* 24th *International Conference on Very Large Data Bases (VLDB)*, Aug. 1998.

[18] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Apr. 2005.

[19] A. Kongmunvattana, S. Tanchatchawal, and N.-F. Tzeng. Coherence-based Coordinated Checkpointing for Software Distributed Shared Memory Systems. In *Proceedings of the* 20th *International Conference on Distributed Computing Systems (ICDCS)*, Apr. 2000.

[20] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux Implementation of a Log-structured File System. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[21] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, June 2007.

[22] S. F. Li, Q. Stafford-Fraser, and A. Hopper. Integrating Synchronous and Asynchronous Collaboration with Virtual Network Computing. *IEEE Internet Computing*, 4(3):26–33, May - June 2000.

[23] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, Apr. 1997.

[24] LVM2 Resource Page. `http://sources.redhat.com/lvm2/`.

[25] J. Nielsen. *Designing Web Usability*. New Riders Publishing, Indianapolis, IN, 2000.

[26] OpenMemex. `http://openmemex.devjavu.com`.

[27] OpenVZ. `http://www.openvz.org`.

[28] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the* 5th *Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[29] T. R. Ostrach. Typing Speed: How Fast is Average, 1997. `http://www.readi.info/TypingSpeed.pdf`.

[30] D. Patterson and J. Gray. A Conversation with Jim Gray. *ACM Queue*, 1(3), June 2003.

[31] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the USENIX Winter 1995 Technical Conference*, Jan. 1995.

[32] J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed Differences: An Algorithm for Fast Incremental Checkpointing. Technical Report CS-95-302, University of Tennessee, Aug. 1995.

[33] J. Pruyne and M. Livny. Managing Checkpoints for Parallel Programs. In *Proceedings of the IPPS Second Workshop on Job Scheduling Strategies for Parallel Processing*, Apr. 1996.

[34] B. K. Schmidt. *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*. PhD thesis, Stanford University, Aug. 2000.

[35] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 2nd edition, 1992.

[36] C. Soules and G. Ganger. Connections: Using Context to Enhance File Search. In *Proceedings of the* 20th *ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[37] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable Remote Execution Facilities for the V-system. In *Proceedings of the* 10th *ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1985.

[38] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, Feb. 1999.

[39] Tsearch2 - full text extension for PostgreSQL. `http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/`.

[40] A. Tucker. Andy Tucker's Blog: What was that again? `http://atucker.typepad.com/blog/2006/11/what_was_that_a.html`.

[41] VMware Movie Capture. `http://www.vmware.com/support/ws5/doc/ws_running_capture.html`.

[42] VMware Workstation 6 Beta. `http://www.vmware.com/products/beta/ws/`.

[43] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the* 6th *Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[44] Windows Desktop Search. `http://desktop.msn.com`.

[45] WinFS. `http://msdn.microsoft.com/data/WinFS/`.

[46] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage*, 2(1), Mar. 2006.

[47] xvidcap. `http://xvidcap.sourceforge.net/`.

[48] Yahoo Desktop Search. `http://desktop.yahoo.com`.

[49] E. R. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the* 11th *ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 1987.