ABSTRACT

A Personal Virtual Computer Recorder

Oren Laadan

Continuing advances in hardware technology have enabled the proliferation of faster, cheaper, and more capable personal computers. Users of all backgrounds rely on their computers to handle ever-expanding information, communication, and computation needs. As users spend more time interacting with their computers, it is becoming increasingly important to archive and later search the knowledge, ideas and information that they have viewed through their computers. However, existing state-of-the-art web and desktop search tools fail to provide a suitable solution, as they focus on static, accessible documents in isolation. Thus, finding the information one has viewed among the ever-increasing and chaotic sea of data available from a computer remains a challenge.

This dissertation introduces DejaView, a personal virtual computer recorder that enhances personal computers with the ability to process display-centric content to help users with all the information they see through their computers. DejaView continuously records a user’s session to provide a complete WYSIWYS (What You Search Is What You’ve Seen) record of a desktop computing experience, enabling users to playback, browse, search, and revive records, making it easier to retrieve and interact with information they have seen before.

DejaView records visual output, checkpoints corresponding application and file system states, and captures onscreen text with contextual information to index the
record. A user can then browse and search the record for any visual information that has been previously displayed on the desktop, and revive and interact with the desktop computing state corresponding to any point in the record. DejaView introduces new, transparent operating system, display and file system virtualization techniques and novel semantic display-centric information recording, and combines them to provide its functionality without any modifications to applications, window systems, or operating system kernels. Our results demonstrate that DejaView can provide continuous low-overhead recording without any user-noticeable performance degradation, and allows users to playback, browse, search, and time-travel back to records fast enough for interactive use.

This dissertation also demonstrates how DejaView’s execution virtualization and recording extend beyond the desktop recorder context. We introduce a coordinated, parallel checkpoint-restart mechanism for distributed applications that minimizes synchronization overhead and uniquely supports complete checkpoint and restart of network state in a transport protocol independent manner, for both reliable and unreliable protocols. We introduce a scalable system that enables significant energy saving by migrating network state and applications off of idle hosts allowing the hosts to enter low-power suspend state, while preserving their network presence. Finally, we show how our techniques can be integrated into a commodity operating system, mainline Linux, thereby allowing the entire operating systems community to benefit from mature checkpoint-restart that is transparent, secure, reliable, efficient, and integral to the Linux kernel.
Contents

1 Introduction 1
   1.1 Display Recording ........................................... 6
   1.2 Content Recording ........................................... 7
   1.3 Execution Recording ........................................ 10
   1.4 Contributions ................................................ 12
   1.5 Organization of this Dissertation .......................... 17

2 System Overview 19
   2.1 Usage Model ................................................. 19
   2.2 Example Scenarios .......................................... 23
      2.2.1 Parking Ticket Proof .................................. 23
4.4 Evaluation ................................................. 67
  4.4.1 Performance Overhead ................................. 70
  4.4.2 Single Application Text Coverage ..................... 71
  4.4.3 Multiple-Application Text Coverage ................... 73
  4.4.4 Tree Characteristics .................................. 74
4.5 Lessons Learned ........................................... 77
4.6 Summary ................................................. 79

5 Virtual Execution Environment ....................... 80
  5.1 Operating System Virtualization ......................... 82
  5.2 Interposition Architecture ............................... 86
  5.3 Virtualization Challenges ................................ 88
    5.3.1 Race Conditions ....................................... 91
      5.3.1.1 Process ID Races .................................. 92
      5.3.1.2 PID Initialization Races ............................ 96
      5.3.1.3 SysV IPC Races .................................... 100
      5.3.1.4 Pseudo Terminals Races ............................ 106
    5.3.2 File System Virtualization ............................ 108
    5.3.3 Pseudo File Systems ................................... 108
  5.4 Evaluation ............................................ 110
    5.4.1 Micro-benchmarks ..................................... 111
    5.4.2 Application Benchmarks ................................ 116
  5.5 Summary ................................................ 120

6 Live Execution Recording ............................... 121
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Whole System Evaluation</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>7.1 System Overhead</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>7.2 Access To Data</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>7.3 Summary</td>
<td>186</td>
</tr>
<tr>
<td>8</td>
<td>Distributed Checkpoint-Restart</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>8.1 Architecture Overview</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>8.2 Distributed Checkpoint-Restart</td>
<td>190</td>
</tr>
<tr>
<td></td>
<td>8.3 Network State Checkpoint-Restart</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>8.4 Evaluation</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td>8.4.1 Virtualization Measurements</td>
<td>209</td>
</tr>
<tr>
<td></td>
<td>8.4.2 Checkpoint-Restart Measurements</td>
<td>209</td>
</tr>
<tr>
<td></td>
<td>8.5 Summary</td>
<td>215</td>
</tr>
<tr>
<td>9</td>
<td>Desktop Power Management</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>9.1 Architecture Overview</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>9.2 Application Containers</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td>9.3 Application Migration</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td>9.3.1 Checkpoint and Restart Overview</td>
<td>228</td>
</tr>
<tr>
<td></td>
<td>9.3.2 Base Connection State</td>
<td>231</td>
</tr>
<tr>
<td></td>
<td>9.3.3 Dynamic Connection State</td>
<td>235</td>
</tr>
<tr>
<td></td>
<td>9.4 Evaluation</td>
<td>238</td>
</tr>
<tr>
<td></td>
<td>9.5 Summary</td>
<td>244</td>
</tr>
<tr>
<td>10</td>
<td>Checkpoint-Restart in Linux</td>
<td>245</td>
</tr>
<tr>
<td></td>
<td>10.1 Usage</td>
<td>246</td>
</tr>
</tbody>
</table>
List of Figures

2.1 DejaView screenshot ................................................. 20
2.2 DejaView architecture ................................................ 30

4.1 Desktop accessibility framework ...................................... 51
4.2 GUI windows and accessibility tree of GEdit ...................... 55
4.3 Capture architecture .................................................. 59
4.4 Capture coverage versus think-time ................................. 72
4.5 Orca coverage versus think-time .................................... 72
4.6 Mirror tree size ....................................................... 75
4.7 Mirror tree memory ................................................... 75
4.8 Query time vs. node size ............................................. 76
4.9 Query time vs. node count ........................................... 76

5.1 Anatomy of virtualization wrappers ................................. 91
5.2 PID deletion race ..................................................... 93
5.3 PID initialization race ............................................... 97
5.4 IPC reuse race ....................................................... 103
5.5 Virtualization cost on UP - micro-benchmarks .................... 112
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Virtualization cost on SMP - micro-benchmarks</td>
<td>115</td>
</tr>
<tr>
<td>5.7</td>
<td>Virtualization cost for LMbench</td>
<td>116</td>
</tr>
<tr>
<td>5.8</td>
<td>Virtualization cost for macro-benchmarks</td>
<td>118</td>
</tr>
<tr>
<td>5.9</td>
<td>Virtualization scalability</td>
<td>119</td>
</tr>
<tr>
<td>6.1</td>
<td>Simple process forest</td>
<td>152</td>
</tr>
<tr>
<td>6.2</td>
<td>Process forest with deletions</td>
<td>153</td>
</tr>
<tr>
<td>6.3</td>
<td>Average checkpoint size</td>
<td>163</td>
</tr>
<tr>
<td>6.4</td>
<td>Average number of processes</td>
<td>164</td>
</tr>
<tr>
<td>6.5</td>
<td>Average checkpoint time</td>
<td>164</td>
</tr>
<tr>
<td>6.6</td>
<td>COW and buffering impact</td>
<td>166</td>
</tr>
<tr>
<td>6.7</td>
<td>Checkpoint time breakdown</td>
<td>167</td>
</tr>
<tr>
<td>6.8</td>
<td>Average restart time</td>
<td>168</td>
</tr>
<tr>
<td>7.1</td>
<td>Recording runtime overhead</td>
<td>176</td>
</tr>
<tr>
<td>7.2</td>
<td>Total checkpoint latency</td>
<td>177</td>
</tr>
<tr>
<td>7.3</td>
<td>Recording storage growth</td>
<td>179</td>
</tr>
<tr>
<td>7.4</td>
<td>Latency of text search and display browsing</td>
<td>182</td>
</tr>
<tr>
<td>7.5</td>
<td>Playback speedup</td>
<td>183</td>
</tr>
<tr>
<td>7.6</td>
<td>Revive latency</td>
<td>184</td>
</tr>
<tr>
<td>8.1</td>
<td>Coordinated checkpoint timeline</td>
<td>194</td>
</tr>
<tr>
<td>8.2</td>
<td>Non-overlapping and overlapping data queues</td>
<td>204</td>
</tr>
<tr>
<td>8.3</td>
<td>Application completion times on vanilla Linux and ZapC</td>
<td>210</td>
</tr>
<tr>
<td>8.4</td>
<td>Distributed application checkpoint times</td>
<td>211</td>
</tr>
<tr>
<td>8.5</td>
<td>Distributed application restart times</td>
<td>212</td>
</tr>
</tbody>
</table>
8.6 Distributed application checkpoint sizes .................................. 213

9.1 NetCONT architecture overview ........................................... 218

9.2 Restore established connections ........................................... 234

9.3 Restore pending connections .............................................. 236

9.4 Average container migration sizes ....................................... 241

9.5 Average container migration times ....................................... 242

10.1 A simple checkpoint-restart example ................................. 250
List of Tables

4.1 A subset of GNOME accessibility roles . . . . . . . . . . . . . . . . . 52
4.2 A subset of GNOME accessibility properties . . . . . . . . . . . . . 53
4.3 A subset of GNOME accessibility events . . . . . . . . . . . . . . . . 54
4.4 Content recording: application scenarios . . . . . . . . . . . . . . . 68
4.5 Summary of benchmark results with think-time of 1000 ms . . . . . 70
4.6 Multiple-application text coverage . . . . . . . . . . . . . . . . . . . 73
5.1 Kernel subsystems and related resources . . . . . . . . . . . . . . . . 85
5.2 Summary of virtualization methods . . . . . . . . . . . . . . . . . . . 89
6.1 Possible flags in the status field . . . . . . . . . . . . . . . . . . . . . 147
6.2 Live execution recording: application scenarios . . . . . . . . . . . 162
6.3 Checkpoint-restart performance for OpenVZ . . . . . . . . . . . . . 169
6.4 Checkpoint-restart performance for Xen VMs . . . . . . . . . . . . 170
7.1 Whole system evaluation: application scenarios . . . . . . . . . . . 174
9.1 Network connections states . . . . . . . . . . . . . . . . . . . . . . . . 232
9.2 Distributed checkpoint: application scenarios . . . . . . . . . . . . 239
9.3 NetCONT power consumption ........................................... 243

10.1 System call flags (checkpoint and restart) ...................... 251
10.2 Kernel API by groups ................................................... 268
10.3 Checkpoint-restart performance .................................... 273
10.4 Checkpoint times and memory sizes ............................... 274
List of Algorithms

5.1 System call wrapper (**getpid**) ........................................... 100
6.1 DumpForest ...................................................................... 148
6.2 FindCreator ..................................................................... 149
6.3 AddPlaceHolder ................................................................. 150
6.4 MakeForest ...................................................................... 154
6.5 ForkChildren ................................................................... 155
6.6 ForkChild ........................................................................ 156
8.1 Coordinated Checkpoint: Manager ........................................... 191
8.2 Coordinated Checkpoint: Agent ............................................. 192
8.3 Coordinated Restart: Manager ................................................. 196
8.4 Coordinated Restart: Agent .................................................. 196
9.1 Network Checkpoint ............................................................. 229
9.2 Network Restart ................................................................. 229
9.3 RestoreEstablishedConnection ............................................. 233
9.4 RestorePendingConnection .................................................. 235
Acknowledgements

This work has seen support from many individuals, and I’d like to use this opportunity to name a few. To begin with, I’d like to thank my advisor, Jason Nieh, to whom I’m deeply indebted. I have enjoyed his knowledge, inspiration, guidance, mentorship, as well as friendship and kind words in times of need. I truly admire Jason’s ability to ask the sharpest of questions that would send me time and again back to the drawing board, often to re-surface with even better ideas; To be able to ask “Jason-grade” questions myself is a source of pride. Working with him has been challenging and rewarding, and having endured his uncompromising demand for quality and relentless attention to details has equipped me with useful skills for the years to come. I’m grateful to have had such a great researcher and person for an advisor.

I wish to also thank Luis Gravano with whom I worked closely on a major section of my thesis. Angelos Keromytis provided feedback and guidance during my time here. This work has additionally benefited from feedback and advice from the rest of my PhD committee members, Gail Kaiser, Dilma da Silva, and Michael Stumm, whom I’d like to thank for their helpful input, comments, and patience.

This dissertation has benefited from many discussions with and contribution from others. In particular, Ricardo Baratto’s research on display virtualization and Shaya
Potter’s ideas on file system unioning were instrumental to this work. Dan Phung made significant contributions to distributed and incremental checkpointing. Ricardo, Dan, and Shaya have provided many insights on this work, implemented parts of the prototype, and participated in its sisyphian evaluation. Stelios Sidiropoulos-Douskos provided excellent advice on research matters and life in general, as well as the occasional supply of superb seafood. Eli Brosh and Christoffer Dall provided invaluable feedback on earlier drafts of this document. These truly exceptional individuals whom I have met throughout my journey at Columbia have not only made this dream possible, but have also been a significant part of my graduate experience. Above everything else, I cherish their friendship. I’d also like to mention Andrew Shu, Nicolas Viennot, Adrian Frei, and Yuly Finkelberg, who contributed ideas, code, comments, and valuable feedback.

Many thanks to the administrative staff in the Computer Science Department for masking out most of Columbia’s red-tape from me. That includes Alice Cueba, Twinkle Edwards, Patricia Hervey, Remiko Moss, Elias Tesfaye, and Cindy Walters. A special thanks to CRF staff, including Daisy Nguyen, Paul Blaer, Shlomo Hershkop, Quy O, and Hi Tae Shin, who worked relentlessly to address my systems needs.

Finally, I wish to thank Franci for her unconditional support and encouragement during this almost-infinite stretch of time, and her incredible ability to keep me a safe distance from insanity, despite my grumpiest moments. Last but not least I’m grateful to my parents and my sisters for their distant yet unequivocal support.

(This work was supported in part by a DOE Early Career Award, AMD, and an IBM SUR Award, NSF grants CNS-0717544, CNS-0914845, CNS-0905246, and ANI-0240525, and AFOSR MURI grant FA9550-07-1-0527.)
...לאה לאיה
Chapter 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. Meanwhile, computers are getting faster at generating, distributing, and storing vast amounts of data, yet humans are not getting any faster at processing it. Exponential improvements in processing, networking, and storage technologies are not making this problem easier. Vannevar Bush’s Memex vision [25] 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, this dissertation introduces a new approach for keeping track of the massive amount of data seen through the desktop, and for being able to playback, browse, search, access, and interact with this data, making it easier for users to retrieve information seen before.
Some systems and tools have been developed to address aspects of the problem of finding information seen on the desktop, including web search engines and desktop file search tools. However, web search engines [176, 177, 178] focus only on static information available on the web. They 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 [69]. Nor do they help with the wealth of additional information that users produce and consume through their personal computers.

Desktop file search tools [43, 44, 45] go a step further and provide mechanisms to search over various forms of individual user documents such as user files, email messages, web pages, and chat sessions. However, while desktop search tools search within current files that may be of interest, they do not return results from files that are no longer available, or from information seen by the user but never actually saved to files.

More importantly, focusing on such individual, relatively static documents in isolation is often insufficient. For a number of important search scenarios, the history and patterns of access to all information on a desktop—static or otherwise—are themselves valuable and, in fact, critical to answer certain queries. For example, suppose a user encounters a problem with her computer. She pieces together a solution through extensive investigation that involves a number of different activities, including manual inspection of files, web search, email search and IM sessions with a few friends. Some time later, the same problem recurs with a similar set of applications running, producing a similar output. Unfortunately, existing search tools provide no way to simply search the user’s past computing experience to identify the user’s previous solution to the problem. All that is possible for the user is to painstakingly attempt to redo the solution through the retrieval of individual documents.
To avoid repeating this time-consuming investigation, what is really needed is a tool that can perform a display-centric search. This tool need not only search across multiple individual resources, but also leverage the rich history of previous interactions with all the information displayed on a desktop. The rationale for this is that the display is the primary means of output of computers: the rich visual display output is a good match for the human visual system, and paramount for the human-computer interaction experience. Displays are becoming bigger and better in their ability to show visual contents to users, and more and more of this information is designed for only human visual consumption and is not available in other ways. Enhancing computers in their ability to process display-centric content will help users with finding information they have seen through their computers.

The concept of desktop-centric search offers several advantages. First, it enables search for data even if it is not explicitly saved (either as a file, bookmark, or note). Second, it provides useful context, such as the provenance of output, which applications were running, and what GUI actions occurred. This context can supplement the user's query in seeking specific records. Third, it reveals information about the persistence of certain output on the display. Persistence data can benefit users interested in when onscreen content appeared or disappeared, and for how long it was displayed. Fourth, desktop-centric searches can leverage information across multiple windows, to either aggregate data from a single application over multiple windows, or correlate data from different windows and different applications. This empowers users to use the rich context they recall having seen onscreen to formulate richer search queries that yield more accurate results.

This dissertation presents 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, enabling users to playback, browse, search, and
revive records, making it easier to retrieve information they have seen before, even when not explicitly saved. Leveraging continued exponential improvements in storage capacity [127], 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’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.

To support its usage model, DejaView provides tools that continuously record both the display and the execution of the user’s desktop computing environment such that the record can be searched, played, and manipulated at a later time. DejaView records three sets of desktop state at all times. First, it records all visual output generated by the desktop to allow users to quickly browse and playback recorder content. Second,
it records all onscreen text as well as associated contextual information and indexes that data to allow users to quickly search and locate recorded content. Third, it records the live execution state including application and file system state of the desktop to allow users to revive their desktop from a previous time.

In providing this functionality, DejaView needs to address a number of challenges along three dimensions, namely usability, performance, and transparency:

- **Usability.** In recording a user’s computing experience, DejaView should ensure that the resulting record is maximally usable by the user. For desktop display, DejaView should preserve the display fidelity, to provide a full fidelity visual experience for playback and browsing. For onscreen contents, DejaView should provide accurate coverage of onscreen data seen by the user, in both foreground and background windows, so that users can search for any content seen before. For live execution, DejaView should save the application state of the multiple processes in a session in a manner that is globally consistent and preserves process dependencies, so that past sessions can be correctly revived. Furthermore, DejaView should allow multiple revived sessions from different points in time to run concurrently without conflicts among each other or with the present session, so that users can interact with multiple sessions at a time.

- **Performance.** Throughout collecting all the data to record, DejaView should be mindful of any overhead it creates on the interactive performance of the desktop, to sustain seamless operation and maintain a pleasant user experience. Furthermore, user actions such as browse, search and session revive should provide acceptable responsiveness for interactive use, allowing DejaView to operate within attention thresholds of end users. Finally, DejaView’s recording should be space efficient, to allow longer periods of recording while consuming less disk
storage a user may wish to use for other purposes, and to potentially reduce access time to records as less data will need to be scanned.

- **Transparency.** For DejaView to be useful in practice, it is crucial that it transparently support the large existing installed base of desktop applications and commodity operating systems, without requiring any modifications to applications, standard libraries, desktop environments, window systems, or operating system kernels.

DejaView transparently provides its functionality, including display, text, and live execution state recording, by introducing lightweight virtualization mechanisms and utilizing available accessibility interfaces. DejaView’s virtualization architecture consists of three main components. For display recording, DejaView virtualizes the display to capture and log low-level display commands, enabling them to be replayed at full fidelity at a later time. For onscreen content recording, DejaView utilizes accessibility interfaces to simultaneously capture displayed text and contextual information to automatically index the display record so it can be searched. For live execution recording, DejaView combines display and operating system virtualization to decouple window system and application state from the underlying system, allowing them to be continuously checkpointed and later revived.

### 1.1 Display Recording

DejaView records the display output to produce a continuous log of all visual information users have had access to through their computers, so that they can be played back, and arbitrarily browsed. Screencasting tools can also take visual screenshots of the display frequently enough to provide such recording, however, incur high record-
ing overhead and storage requirements. Using lossy MPEG or JPEG encoding could be attempted to reduce the storage needs, but this would further increase recording overhead and decrease the quality of the display record.

Instead, DejaView uses a virtual display architecture that 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’s virtual display architecture leverages the standard video driver interface to introduce a virtual display driver that intercepts drawing commands, which makes recording and playback simple. DejaView takes advantage of this mechanism to record and display simultaneously. As visual output is generated, the virtual display driver multiplexes the output into commands for the display, and commands for logging to persistent storage.

DejaView’s virtual display architecture has three key benefits in terms of transparency, usability, and performance. By using a standard interface, DejaView can operate seamlessly with existing unmodified applications, window systems and operating systems. Because it intercepts and captures all low level display commands, DejaView produces a complete, lossless display recording that can be replayed later with full fidelity. Lastly, working at the level of display commands allows DejaView to record only display updates rather than full screenshots to minimize the storage requirements.

1.2 Content Recording

In addition to visual output, DejaView records onscreen text and contextual information to automatically index the display record so it can be searched. Contextual information includes data such as the window from which the text came from, the duration in which it was on the screen, etc. Capturing textual information from display
commands is often not possible due the variety of application-specific mechanisms for rendering text. Screen capture tools can take a visual snapshot of the display, but the resulting snapshot is simply a set of pixels with no semantic information. While optical character recognition (OCR) could be used to extract text from screenshots, this is at best a very slow and inaccurate process that cannot support real-time capture of onscreen content.

Instead, DejaView leverages ubiquitous accessibility infrastructure available on modern operating systems and widely used by screen readers to provide desktop access for visually-impaired users [1]. This infrastructure is 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 a hyperlink).

Unlike screen readers which are window-centric and limited to watching changes to a foreground window only, DejaView is desktop-centric and must track multiple applications to record the onscreen contents of the desktop as a whole. This involves extracting significantly more data than traditional screen readers. The accessibility infrastructure, however, relies on message passing for desktop applications to expose their onscreen contents, and is sensitive to communication latencies between processes. Screen readers typically tolerate query latencies because they target only the window in focus, issue few queries, and need not perform faster than at interactive rates. Conversely, retrieving the full accessibility data of a whole desktop comprising of multiple applications could generate a multitude of queries, which would become prohibitively expensive. A typical scan would not only generate an excessive runtime load on the system, but also produce inaccurate and outdated results because the
accessibility data may change in the interim, and may impair the system’s responsiveness during query bursts.

To address this difficulty, we introduce a novel display-centric text recorder that facilitates real-time access to both foreground and background onscreen text with low overhead. The recorder provides an intelligent caching architecture that integrates with the accessibility infrastructure to reduce the need for accessibility queries. On-screen data as well as the structure of the onscreen data is efficiently cached, making available not just the results of one query but a complete view of all onscreen data at a given time. DejaView leverages this caching architecture to continuously log the onscreen contents into structured text documents whose contents reflect the text and associated contextual information at particular moments. These documents carry valuable data about the display structure that can be used to deliver better search results to the user.

DejaView’s content recording architecture has three key benefits in terms of transparency, usability, and performance. By using a mechanism natively supported by applications, DejaView can operate transparently with existing unmodified applications, windows systems, and desktop environments. By leveraging the accessibility mechanisms, DejaView has maximum access to all display-centric content including data and metadata associated with both foreground and background windows. Lastly, using a caching architecture allows DejaView to mitigate the high performance costs of using the accessibility infrastructure while retaining its full functionality for accessing onscreen data and providing an accurate mirror of this data available for use in real-time.
1.3 Execution Recording

For cases where only visual information such as screenshots and text snippets are not enough, DejaView continuously records an entire live user session such that it can be revived later to allow users to interact with their desktop as it was at any point in the past. Virtual machine monitors (VMMs) could be used to transparently checkpoint and later roll back an entire operating system environment, consisting of both the operating system and the applications. However, because they operate on entire operating system instances, they incur visible runtime overhead and prohibitive checkpoint-restart overheads. VMMs could also be used to log entire operating instances and their applications and later replay their execution. However, implementing deterministic replay without incurring prohibitive overhead remains a difficult problem, especially for multiprocessor systems. Moreover, it would require re-executing everything for playback, which is unsuitable for interactive use.

Instead, DejaView leverages operating system virtualization mechanisms to introduce a lightweight virtual execution environment that decouples the user’s desktop computing session from the underlying operating system instance, enabling it to continuously checkpoint an entire live user session and later revive the session in a consistent state from any checkpoint. This lightweight virtualization imposes negligible overhead as it operates above the operating system instance to encapsulate only the user’s desktop computing session, as opposed to an entire machine instance.

The virtual execution environment transparently encapsulates the user’s desktop computing session in a private virtual namespace. By providing a virtual namespace, revived sessions can appear to access the same operating system resources as before even if they are remapped to different underlying operating system resources upon revival. By providing a private namespace for each session, revived sessions from
different points in time can run concurrently and appear to use the same operating system resources inside their respective namespaces without any resource conflicts across sessions.

Building on this virtualization, DejaView records the user’s desktop session by frequently checkpointing all the operating system state associated with all the processes in the session. Since checkpoints record not just a single process, but an entire session consisting of multiple processes and threads, the application state saved and restored must be globally consistent. We introduce a novel algorithm for accounting for process relationships that correctly saves and restores all process state in a globally consistent manner. This algorithm is crucial for enabling transparent checkpoint-restart of interactive graphical applications. We also introduce an efficient algorithm for identifying and accounting for shared resources and correctly saving and restoring such shared state across cooperating processes. 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.

Unlike hardware virtualization, where each virtual machine has a full operating system instance, DejaView only saves user desktop state, not the entire operating system instance. Checkpointing at this finer granularity is crucial to reduce the amount of state to be saved and the extent of each checkpoint, as well as the response time to revive a past session. To further reduce the application downtime incurred by checkpoints, DejaView employs various optimizations including shifting expensive I/O operations out of the critical path, and using fast incremental and copy-on-write techniques. Checkpoint policies that decide how frequently to checkpoint based on user and display activity further reduce noticeable impact as well as storage requirements.
DejaView execution recording architecture has three key benefits in terms of transparency, performance, and usability. By encapsulating a user’s desktop in a virtual execution environment based on a kernel module, DejaView can operate seamlessly with existing unmodified applications and operating system kernels. This encapsulation also allows DejaView to revive past sessions so that a user can interact with them and enables multiple revived sessions to run concurrently without conflicts among each other or with the present session. Lastly, checkpointing at a fine granularity and optimizing to reduce the frequency and the duration of checkpoints allow DejaView to minimize any impact on interactive desktop application performance and reduce the storage requirements.

1.4 Contributions

This dissertation presents DejaView, 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. 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. More specifically, novel contributions of this dissertation include:

1. We introduce a lightweight virtual execution environment architecture [96] that decouples applications from the underlying operating system. It transparently encapsulates processes in isolated containers using a lightweight virtualization layer between the applications and the operating system, without any underlying operating system kernel changes.
2. We discuss key implementation issues and challenges in providing operating system virtualization in a commodity operating system. We compare methods for implementing virtualization at user-level vs. in-kernel, discuss performance costs for methods to store virtualization state, and analyze subtle race conditions that can arise. The experiences from this approach are instrumental in demonstrating how operating system virtualization can be incorporated into commodity operating systems with minimal changes and low overhead.

3. We implement an operating system virtualization prototype entirely in a loadable kernel module that works across multiple Linux kernel versions, demonstrating the portability of our approach. We present qualitative results showing that our minimally invasive approach can be done with very low overhead.

4. We introduce a transparent checkpoint-restart mechanism for commodity operating systems [95] that can checkpoint multiple processes in a consistent state and later restart them. The approach combines a kernel-level checkpoint mechanism with a hybrid user-level and kernel-level restart mechanism to leverage existing operating system interfaces and functionality as much as possible.

5. We introduce a novel 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.

6. We introduce a novel algorithm to account for process relationships that correctly saves and restores all process state in a globally consistent manner, and an efficient algorithm to identify and account for shared resources, to correctly save and restore such shared state across cooperating processes.
7. We introduce several optimizations to reduce application downtime during checkpoints. Some optimizations shift the latency of expensive I/O operations before and after the application downtime. Others reduce the amount of state needed to be saved during application downtime. These are crucial to allow frequent checkpoints without any noticeable performance degradation.

8. We implement an application checkpoint-restart mechanism as a loadable kernel module and userspace utilities in Linux. We demonstrate its ability to provide transparent checkpoint-restart functionality on real-world applications without modifying existing system components.

9. We introduce Capture, a display-centric text recorder that facilitates real-time access to onscreen contents and its structure and contextual information, including data associated with both foreground and background windows. The recorder makes novel use of the accessibility infrastructure available on modern operating systems to continuously track onscreen text and metadata, without any changes to applications or window systems. Recorded data can benefit a variety of problem domains, including assistive technologies, desktop search, auditing, and predictive graphical user interfaces.

10. We introduce an intelligent caching architecture that reduces the need to query the accessibility infrastructure for onscreen data using a pull model in which multiple screen update notifications from the accessibility infrastructure are coalesced to be handled by a single query. The architecture mitigates the high performance costs of using the accessibility infrastructure while retaining its full functionality for accessing onscreen data, and makes available not just the results of one query, but a complete view of all onscreen data at a given time.
11. We implement a *Capture* prototype without any changes to applications or windows systems. We demonstrate the accuracy and efficacy of the prototype with a wide range of desktop applications that generate updates to onscreen data at a high frequency. Compared to a standard screen reader, it records substantially higher percentage of onscreen text for both foreground and background windows with textual content, and it records many common application workloads not handled by the screen reader.

12. We introduce DejaView [93], a personal virtual computer recorder architecture that records a user’s session efficiently, without user-perceived degradation of application performance. We combine display, operating system, and file system virtualization to provide this functionality transparently without any modifications to applications, window systems, or operating system kernels.

13. We introduce a policy for throttling of checkpoints to minimize both the runtime overhead due to checkpoints and the storage requirements. The policy reduces runtime overhead by limiting checkpoints rate. It reduces storage requirements by employing optimizations that skip checkpoints in the absence of display updates or when display activity is low.

14. We implement a DejaView prototype and evaluate its performance on common desktop application workloads and with real desktop usage. We show that recording adds negligible overhead, capturing the display and execution state of interactive applications with only a few milliseconds of interruption. We show that playback can enable users to quickly view display records much faster than real-time, and that browsing and searching display information is fast enough for interactive use.
The individual components that build up DejaView provide key technologies useful in a broader context beyond the desktop recorder framework. This dissertation also demonstrates the applicability of providing application mobility:

15. We introduce ZapC [97], a transparent coordinated checkpoint-restart of distributed networked applications on commodity clusters. ZapC can checkpoint an entire distributed application across all nodes in a coordinated manner such that the application can be restarted from the checkpoint on a different set of cluster nodes at a later time. Checkpoint and restart operations execute in parallel across different nodes. Network state, including socket and protocol state for both TCP and UDP, is saved in a transport protocol independent manner.

16. We introduce NetCONT, a system that enables energy savings by allowing idle hosts to transition to low-power state while preserving their network presence even when they sleep. NetCONT seamlessly migrates network applications and their existing connections from idle hosts preparing to sleep to a dedicated server where they continue to run unmodified, and relies on the applications themselves to maintain their network presence. Migrating individual applications provides good scalability to consolidate applications from hundreds of hosts into a single server, and allows fast migration times that do not impact the user experience.

17. We present Linux-CR [94], an in-kernel implementation of transparent application checkpoint-restart aiming for the Linux mainline kernel. Building on the experience garnered through DejaView and on recent support for virtualization available in mainline Linux, Linux-CR’s checkpoint-restart is transparent, secure, reliable, efficient, and well integrated with the Linux kernel.

Finally, these technologies further benefit additional research both directly and indirectly. For example, the namespace virtualization ideas of our virtual execution
environment are now part of the Linux kernel [19]. Explicitly building on our virtualization, \textsc{Scribe} [98] employs lightweight operating system mechanisms to provide deterministic application execution-replay on commodity multiprocessor operating systems. \textsc{MediaPod} [133] and \textsc{GamePod} [134] are portable systems that enable mobile users to maintain the same persistent, personalized environments, for multimedia and gaming respectively, using a mobile storage device that contains complete application-specific environments. They build on our virtualization and checkpoint-restart mechanisms to decouple a desktop environment and applications from the host, enabling a user’s session to be suspended to the device, carried around and resumed on another computer. \textsc{ASSURE} [152, 153] is a system for automatic self healing of software systems to enhance security and robustness, that takes advantage of a variant of our checkpoint-restart mechanism that provides ultra-fast in-memory checkpoint and rollback capabilities to reach good performance levels. Details on these are beyond the scope of this dissertation.


e1.5 Organization of this Dissertation

This dissertation is organized as follows. Chapter 2 provides a general overview of \textsc{DejaView}, the usage model and scenarios, and the overall architecture. Chapter 3 presents \textsc{DejaView}’s display recording architecture and the mechanisms available to access the recorded information. Chapter 4 presents \textsc{DejaView}’s display-centric content recording architecture and its use of the accessibility infrastructure. Chapter 5 describes \textsc{DejaView}’s virtual execution environment, and Chapter 6 describes \textsc{DejaView}’s live execution recording architecture and the mechanisms available to revive a past session from recorded data. Chapter 7 combines the three recording mechanisms together to provide a whole system evaluation of \textsc{DejaView}. Chapters 8–10
describe three systems that leverage DejaView’s architectural building blocks: distributed checkpoint-restart, desktop power management, and checkpoint-restart in Linux, respectively. Chapter 11 discusses related work. Finally, we present some conclusions and directions for future work in Chapter 12.
Chapter 2

System Overview

Before describing the technology behind DejaView, in this chapter we begin with a general overview of the system and how it is used. First, we present a description of the usage model of DejaView to provide an understanding of how users interact with the system. Next, we discuss a number of concrete examples that illustrate the usefulness of DejaView’s unique functionality in real-life scenarios. We then present a high-level overview of the system’s architecture and how it supports all of its functionality.

2.1 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 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
Figure 2.1 – A 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 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 GUI widgets to access DejaView’s recording functionality, shown in Figure 2.1. 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 tradeoff 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 [27], 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. This prevents the recording of passwords entered by the user. Standard encryption techniques can also be used to provide an additional layer of protection.

From a privacy perspective, DejaView’s default usage model is solipsistic, i.e., hoarding information about oneself for one’s own purposes, thereby rendering privacy
only a minor concern. With the growth of cloud-based services, one can envision storing DejaView’s digital record in the cloud, or even providing its functionality for standard remote desktop services. In this model, the user’s data is no longer in her possession, and becomes more vulnerable to breach of security or trust. For example, governments have power to insist that information that exists is made available to them. Nevertheless, the additional risk is comparable to the privacy risk associated with standard web services such as email, password management, and backup tools, which manage sensitive personal data and have already become ubiquitous.

Recording a user’s desktop experience can be viewed as a form of lifelogging, the undiscriminating collection of information concerning one’s life and behavior [126]. However, while early lifelogging experiments focused on private use, this is no longer the case today. The rapid rise of social networking practices, where users voluntarily generate, and share information with others through personal blogs and web services such as YouTube, Flickr, and Facebook, continuously blurs the boundary between what is private and what is not. Social networking, however, differs from lifelogging in that users actively choose the content to share and the audience to share it with, revealing as much or as little information about themselves as they care to post. Similarly, DejaView could be enhanced with an interface to allow better control over its recording. For instance, it could allow users to stop and resume recording, or discard records, similarly to how journalists choose to go “off the record”. It could also allow users to select what part of the recorded information may become public.

In an information-intensive age where the surrender of digital identity is a commonplace, for purposes such as commerce, marketing, social networking, or receipt of services, personal knowledge management is an issue for anyone who uses digital technologies. Addressing the larger privacy and security ramifications of DejaView’s computing model is beyond the scope of this dissertation.
2.2 Example Scenarios

DejaView’s usage model goes beyond traditional desktop and web search tools in several facets. First, it enables search for data even if it is not explicitly saved. Second, it provides useful context that can supplement the user’s query in seeking specific records. Third, it provides data about the persistence of certain output on the display. Fourth, it can correlate information across multiple windows. DejaView goes beyond screencasting in that it provides the ability to revive and interact with a session instead of only viewing a playback of the display record.

These capabilities create a qualitative advantage for DejaView over existing tools. For example, desktop search tools can examine information in individual files (and therefore per application), but cannot tell which parts of a file’s contents appeared onscreen and at what times. Nor do they integrate information from multiple sources. A multi-term query where each term is found exclusively in a single file would fail to provide relevant matches with these tools, but will succeed under DejaView. The following scenarios illustrate these concepts using concrete examples.

2.2.1 Parking Ticket Proof

Consider a person who is paying a parking ticket online through a Department of Parking Violations website. The user pays the parking ticket right before it is due by credit card, completes the process, and does not bother to save or print the confirmation page that appears at the end of the transaction. A few weeks later, the user receives a notice from the Department of Parking Violations indicating that the ticket was not paid and a penalty fee has been assessed. The user is certain she paid the parking ticket, but realizes she has no written confirmation to show this because she did not save the transaction confirmation page, which is no longer accessible. She
cannot search her desktop files since no file exists corresponding to the transaction. She checks her credit card statement expecting to provide proof using the statement, but the transaction does not appear on her credit card. It appears that there was an error that occurred at some point during the transaction such that it was not properly accounted for by either the Department of Parking Violations or the credit card company, with the former being most likely. A file-based desktop search tool allows users to search persistent files, email messages, etc., but not the transient information that would be required to resolve the ticket dispute discussed above. In contrast, a search using DejaView incorporates the recorded screen data, and would satisfy the information requirement.

2.2.2 Software Re-install

Users occasionally uninstall and reinstall complex software components for various reasons, e.g., for upgrades or due to mis-configurations. Consider a user who installed and configured a web server to host his personal website. Oftentimes such an install requires significant time and effort, particularly if done for the first time or infrequently. A couple of months later, the web server is hacked. The user has no other choice but to reinstall the entire server from scratch in order to ensure that the website remains untainted. During this second install, he faces an error message that he had seen already before. More specifically, he remembers that the solution involved reading the web server manual, browsing several online resources, and editing certain configuration files. Unfortunately, repeating the same steps in search of the solution as before is unlikely to be faster; a web search would yield the same results as previously, and a file-based search tool is of little help. The user can use DejaView to search for the error message, e.g., by searching for when the error message
disappeared from the display, to get to the point in time when he had dealt with the error before, browse the display record to right after the issue had been fixed, and then revive the session and have access to a version of the configuration files that is known to work.

### 2.2.3 Objects Not Indexable

Certain classes of desktop objects carry information whose encoding is other than textual, and are therefore not easily (or not at all) indexable. Common examples include most forms of multimedia files, including image files, audio files and video files. To search for such objects, traditional search techniques generally rely on annotations. In the absence of such annotations, the only way to retrieve such data objects is by using the context from the time they appeared onscreen.

Consider a student who is a fan of the comic strip “PhD Comics,” and frequently exchanges links to preferred strips with her fellow PhD students. In preparation for her thesis defense presentation, she wants to retrieve a specific strip. If the strip was not saved under an explicitly meaningful name, and the strip’s URL is obfuscated (as often is the case), then a file-based search is unlikely to produce useful results. However, with DejaView the student can combine multiple contextual hints that she may remember from that time, such as who mentioned that strip to her and what she was working on at the time, into a single query that exploits onscreen context from multiple windows. Furthermore, once found, the old session may be revived to recover a pristine version of that comic strip.
2.2.4 Searching for Generic Terms

Searching for information using generic search terms is challenging as it requires digging up a few relevant items in a sea of results from a broad range of domains. Adding contextual information to such queries can make them more selective and narrow their scope to the desired breadth.

Consider a professor who loves to try out new restaurants with his wife frequently, particularly on special occasions such as family birthdays and anniversaries. Every year he works hard to select the best restaurant for their anniversary, and usually starts searching for restaurants four weeks before the anniversary. This year, he would like to surpass the expectations set by last year’s selection. He remembers that last year he had researched extensively on the web, and sought recommendations from several friends. He finally had been suggested a fantastic restaurant on a foodie forum, but chose not to go there after having checked its location in Google Maps. It is getting close to the anniversary, and he would like to give that restaurant a chance. Therefore, he wants to retrieve the name of the restaurant. Unfortunately, searching for the term “restaurant” is likely to generate a slew of results even if confined to only his email records, IM records, or browser history. Instead, adding context by searching for multiple terms, such as “restaurant,” “forum,” and “google maps,” and limiting the search to the two-week period before last year’s anniversary is likely to produce a more manageable set of results.

2.2.5 Sysadmin Troubleshooting

To complete certain tasks, users routinely perform on their desktops complex actions that involve multiple applications and windows, often based on ad-hoc decisions. When faced again with a task that they have completed before, users might need to
recreate from scratch their past actions, relying on their memory and on poor (or non-existing) documentation. Thus, users are often forced to waste an unnecessarily long time rediscovering their previous actions, even in the presence of state-of-the-art desktop search tools.

Consider a graduate student who is in charge of upgrading the Linux installation on one of her research group’s computers. During this upgrade, this student runs into an unusual, hard-to-solve problem. The student spends several hours piecing together a solution through extensive investigation that involves a number of different resources, including manual inspection of files, use of automatic utilities, web search, email search, and IM sessions with colleagues. A few weeks later, the student is asked to upgrade the Linux installation on another machine, and encounters the same problem as before. She has not kept proper documentation of its complex solution, so she will have to spend, once again, several hours tracing her earlier steps. A file-based desktop search tool allows the student to search through individual files, email messages, etc., in isolation, but this is insufficient for the student to rediscover the sequence of multi-window, multi-application actions that eventually led to her solving the problem.

2.3 Architecture Overview

To support its personal virtual computer recorder usage model, DejaView needs to record both the display and the 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 three
main components: a virtual display, a display-centric onscreen contents recorder, and a virtual execution environment. These components leverage existing system interfaces to provide transparent operation without modifying, recompiling, or relinking applications, window systems, or operating system 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. 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 display-centric text recorder facilitates real-time access to full onscreen contents, enabling the entire contents seen through the display to be continuously recorded, indexed, and later searched. DejaView leverages the ubiquitous accessibility infrastructure (used by screen readers to provide desktop access for visually-impaired users) to provide an intelligent caching architecture that continuously tracks onscreen contents. The cache is updated using a pull model in which multiple screen update notifications from the accessibility infrastructure are coalesced and handled by a single query back to the accessibility infrastructure. This caching architecture is essential to be able to record with low overhead and not interfere with the user’s interactive experience while simultaneously achieving accurate coverage of all onscreen
data presented to the user. It makes available a complete view of all onscreen data at a given time, including data and metadata associated with both foreground and background windows. Furthermore, by using a mechanism natively supported by applications, DejaView has maximum access to onscreen textual information without requiring any application or desktop environment modifications.

DejaView’s virtual execution environment decouples the user’s desktop computing environment from the underlying operating system, enabling an entire live desktop session to be continuously checkpointed and later revived from any checkpoint. DejaView leverages the standard interface between applications and the operating system 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 operating system resource names as used before being checkpointed, even if they are mapped to different underlying operating system resources upon revival. By providing a private namespace, revived sessions from different points in time can run concurrently and use the same operating system resource names inside their respective namespaces, yet not conflict among each other. This lightweight virtualization mechanism imposes low overhead as it operates above the operating system 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
view, manipulate, and interact with this recorded state. Three 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 onscreen text and associated contextual information including data from both foreground and background windows, which allows users to quickly search and locate recorded content. The third 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.

Figure 2.2 summarizes the main components of DejaView. The figure depicts an overview of the system architecture showing the user’s desktop session decoupled from the operating system through a thin virtualization layer. The session consists of regular desktop applications, and a display server. The display, the onscreen contents and the execution state are continuously recorded to permanent storage, and the text
is indexed. Revived sessions can coexist with the present session, each isolated in a separate virtual execution environment. The user interacts with any desktop session through a viewer that connects to the display server. The viewer can also access the recorded data to perform actions such as search, browse, and playback.

Recording of visual output generated on the desktop is crucial to DejaView’s operation. Although execution recording can re-generate any past state, including the display state, it cannot obviate the explicit visual output recording for two reasons. First, it is only possible to revive sessions from discrete points in time in which checkpoints were taken. Therefore, one cannot guarantee that the state in between two consecutive checkpoints will be accurately regenerated. (Note, however, that this can be rectified by adding deterministic record and replay capabilities [98].) More importantly, using revived sessions cannot produce the necessary state for operations such as browse and playback fast enough for interactive use.
Chapter 3

Display Recording and Search

In this chapter, we present DejaView’s display-centric recording architecture as well as the mechanisms available to access the recorded information. We give an overview of how visual output is recorded using display virtualization that decouples the display state from the underlying hardware, making it easy to manipulate and record. The generated visual record can be browsed and replayed at full fidelity at a later time. Then, we describe how onscreen contents are recorded using a display-centric text recorder that leverages the accessibility infrastructure available on modern operating systems to obtain onscreen text and associated contextual data. DejaView continuously logs the data in a database as records whose contents reflect the onscreen text and contextual information at particular moments. The database is enhanced with text search capabilities, making it easy to index and search the data. We also describe an alternative approach that instead stores the data in structured text documents and then leverages mature desktop search engine technologies to index and later answer search queries about the data. Using such text-shots not only provides more flexibility over databases which require fixes schemas, but also preserves the valuable contextual information about the saved text to be used for better information retrieval.
3.1 Display Recording

To record the display, DejaView virtualizes the display to capture and log low-level display commands, enabling them to be replayed at full fidelity at a later time. DejaView leverages previous work on the THINC [14, 15] 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.
3.1.1 Record

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
Chapter 3. Display Recording and Search

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 § 3.1.2.

DejaView uses three types of files to store the recorded display output: timeline, screenshot, and command. All three types of files are written to in an append-only manner, ensuring that the records are always ordered by time. This organization speeds up both recording and playback. While recording, DejaView does not incur any seeking overhead. During playback, binary search can be used on the index file to quickly locate the records of interest.

A timeline file contains all the meta information required for playback. This file is a collection of tuples of the form \([\text{time}, \text{screenshot}, \text{command}]\), where each tuple represents a point in the timeline where a screenshot was taken, and can be used to start playback. The command component represents the next command that was recorded after the screenshot was taken. Both screenshot and command are tuples of the form \([\text{filename, file_position}]\), and represent pointers to where the actual data for the screenshot and command is stored: the filename of the appropriate file, and the offset within that file where the information is stored.
Screenshot files hold the actual screenshot data. They are organized as a contiguous set of records, where each record is a tuple of the form \([\text{type, time, size, dimensions, data}]\). \text{type} specifies whether the record is a screenshot or a reference to the next screenshot file. \text{time} specifies the time at which the screenshot was recorded. \text{size} specifies the data size of the screenshot. \text{dimensions} specifies the dimensions of the screenshot, to allow for changes of the display’s geometry to be appropriately recorded. \text{data} is the actual screenshot data.

Command files contain the stream of display commands. In the same manner as screenshot files, each command is stored as a serialized record of the form \([\text{type, time, size, data}]\). \text{type} specifies the type of THINC display command. \text{time} specifies the time at which the command was recorded. \text{size} specifies the data size of the command. \text{data} is the actual command data.

DejaView allows for multiple screenshot and command files to be used if needed or desired, for example for systems with maximum file sizes that could be exceeded by long-running desktop recording sessions. A simple mechanism is used to concatenate these files into a continuous logical stream. At the end of each file, a special record is appended that points to the next file on the stream. The record has the same format as other records. It uses the \text{type} field to mark itself as an end-of-file/next-file marker, and the \text{data} component to store the next filename. As playback occurs, this record is read just like any other record, but causes the playback program to start reading from the next file and continue its operation.

### 3.1.2 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.

### 3.2 Content Recording and Search

In addition to visual output, DejaView records onscreen text and associated 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 onscreen text, the window that it came from, the duration in which the text appeared on the screen, etc.

Given the need to capture and process display-centric content, various screen capture tools have been developed to attempt to provide this functionality. They can take a visual snapshot or movie of the display or portions of the display, which can be saved to storage for later viewing. However, the resulting snapshot is simply a set of pixels with no semantic information. For example, any notion of text that originally appeared on the screen is lost, making it impossible to cut, paste, search, or otherwise manipulate textual content that can provide semantic meaning.

Furthermore, because there is a wide array of application-specific mechanisms used for rendering text, capturing textual information from display commands is often not possible. Optical character recognition (OCR) could be attempted to extract such information from the saved screenshots, but this is at best a very slow and inaccurate process that cannot support real-time capture of onscreen content.
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 [1]. 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.

### 3.2.1 Text Capture and Indexing

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 share the functional requirement of capturing textual content off of the display. However, while screen readers interact directly with the accessibility infrastructure, DejaView collects the textual content through Capture, a display-centric text recorder that facilitates real-time access to all onscreen text and associated contextual data. We discuss Capture in detail in the subsequent chapter, Chapter 4.

At startup time, the daemon registers with Capture and asks it to deliver notifications when new text is displayed or existing text on the screen changes. As notifications are received in response to changes in the displayed contents and state, the daemon wakes up, collects the new text and state, and inserts a full snapshot of all the onscreen text into the database. Each row in the database records time-stamped
text from a single application and its contextual data, such that a sequence of rows with a common timestamp compose the entire snapshot. In this manner, DejaView can efficiently search for text that appeared within specific applications, as well as determine the origin of text occurrences in search results.

By storing and indexing the full state of the desktop’s text over time rather than only new or modified text as changes occur, 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 stored and indexed only when it first appeared on the screen, but not thereafter when additional text appears, temporal relationship between the web page and the paper would never have been recorded (each would be indexed separately with distinct timestamps), and the user would be unable to find 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 this 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.
Additionally, DejaView can be enhanced to rely on mature OCR technology on the display recording to derive text from the (increasingly rare) applications for which the accessibility interfaces are either non-existing or unreliable. Using OCR can also benefit display objects whose proper encoding is graphical rather than textual. In this way, DejaView could extract textual information from display objects, such as images in formats like JPEG, PNG or EXIF, that may represent photos of street signs, cartoons with word balloons, or scanned documents. Actual OCR processing can be done in the background or deferred to the night when the computer is idle in an offline fashion to minimize its performance impact on user activity.

### 3.2.2 Search with Database

In addition to standard PVR-like functionality, DejaView provides a mechanism that allows users to quickly search recorded display output. Unlike state-of-the-art desktop search tools (e.g., [9, 43, 44, 45]), DejaView does not retrieve individual files such as user files, email messages, web pages, or chat sessions. Instead, a DejaView keyword search returns full desktop snapshots, each reflecting the state of the desktop at some point in time and including all application windows and their associated text and other relevant features.

DejaView’s search uses the database and 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 in the database, 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 dissertation.

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 in the database. 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 (i.e., 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. Screenshot generation is very
similar to the visual playback described in § 3.1.2, 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 [60], 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.

3.2.3 Search with Text-shots

Using a database augmented with text search capabilities is a first step toward an
efficient storage and indexing of the displayed text. However, the approach has two
shortcomings. First, the database schema defines the set of contextual properties
stored with text records, and it must be chosen in advance, but contextual data is
diverse and quite often application-specific, making it difficult to formulate a schema
comprehensive enough to cover all the possibilities. Moreover, because the schema is
inflexible, it is impossible to extend the contextual data it embodies in response to
new features that appear as accessibility interfaces continue to improve.

More importantly, application text typically comes from the multiple GUI com-
ponents that compose the application windows, including menu items, window titles,
document headings, and text paragraphs, each component associated with its own
contextual data. Grouping the entire displayed text of an application into a single text record in the database disposes of much of the contextual information associated with that text. Only “global” application data is preserved, e.g., window location and dimension and whether it has the focus, while valuable information such as the role of text components and whether some text was selected (highlighted) is completely lost.

To address these limitations and support DejaView’s search model that returns full desktop snapshots, in an efficient manner, DejaView can be enhanced to leverage mature desktop search engine technologies such as Indri [78] and Xapian [184]. For example, we can consider each snapshot as a single text document for indexing, by simply “gluing together” the text extracted from all the application windows associated with the snapshot. We can then apply any state-of-the-art information retrieval strategy [110] to answer keyword queries over these “documents.”

Unfortunately, this scheme is likely to produce suboptimal query results in many cases, because it ignores the display structure on which the extracted text occurred. In particular, this method of text aggregation effectively strips out the rich contextual information obtained via the accessibility interfaces about the text. For example, this scheme will not detect whether query keywords co-occurred in a single application window or were scattered across the display, which may be valuable cues to inform the search ranking strategy.

Instead, we can aggregate the onscreen text into the documents in a manner that preserves the structure of the contents. Specifically, the text documents associated with the desktop snapshots can be formatted using XML, so that the desktop context and structure can be captured flexibly and in turn used for varying ranking strategies. The use of XML offers two key benefits. First, it is flexible and can evolve to accommodate new forms of contextual data exposed by the accessibility interfaces.
Second, it provides natural tagging to the data that is encapsulated, which can be used for indexing and ranking strategies. For example, tags could indicate whether a given text came from a menu item or from an input box, and the text from different tags can be ranked differently, according to their expected relevance.

In this way, DejaView will log the onscreen contents into time stamped text-shots, which are structured text documents whose contents reflect the onscreen text and associated contextual information at particular moments. By keeping the structure, information will be recorded as it was displayed with the same personal context and display layout. It could then be indexed based on displayed text and contextual information captured in the same context as the recorded display data. Therefore, text-shots carry valuable data about the display structure that can be used to deliver better search results to the user. Further exploration of these ideas is beyond the scope of this dissertation and is an interesting topic for future work.

3.3 Summary

In this chapter, we presented DejaView’s display-centric recording which records all viewed output with the same personal context and display layout so that it can be browsed and replayed, and indexes this record based on displayed text and contextual data captured in the same context, so that it can be searched effectively. Visual display recording utilizes a virtual display architecture to duplicate the visual output into a stream for display and a stream for logging to persistent storage, and provides transparent recording with very low storage requirements and performance impact. Onscreen context recording utilizes existing accessibility interfaces to simultaneously capture displayed text and contextual information, which we discuss in depth in the next chapter. We presented two approaches for indexing and searching of onscreen
information. The first approach uses a database enhanced with text search capabili-
ties, making it easy to index and search the data. The second approach stores data
in structured text documents called text-shots and leverages mature desktop search
engine technology to search these documents. Both approaches track the full state of
onscreen content over time rather than only new or modified text, to provide access
to temporal relationships and state transitions of all textual content.
In this chapter, we introduce DejaView’s display-centric content recording architecture and how it uses the accessibility framework to obtain onscreen textual and contextual information, making it easy to index the visual display record so that it can be effectively searched later. Screen readers, too, use accessibility tools to access onscreen data, but are limited to a foreground window only and would produce intolerable overhead were they to continuously track the entire onscreen contents. We present a novel display-centric content recorder that facilitates real-time access to all onscreen data based on an intelligent caching architecture that integrates with the standard accessibility framework. This enables fast, semantic information recording without any modifications to existing system components. We have implemented a prototype of the content recorder and present quantitative results of our prototype on real desktop applications in terms of runtime overhead and accuracy of capturing onscreen data. We show that our prototype imposes only modest performance overhead on a wide range of desktop applications while being able to effectively record all onscreen textual data even when applications are generating updates to onscreen data at a high frequency.
4.1 Display-Centric Text Recording

Computer display are the primary means of output for users. The rich visual display output of computers, which has only become bigger and better over time in its ability to show visual content to users, is a good match for the human visual system and has vastly improved the human-computer interaction experience. However, this form of output is, unfortunately, not a good match for enabling computers to process the information content. The ability of computers to meaningfully do so has lagged far behind the rate at which such information is being generated. This mismatch is a growing problem, as more and more information is designed for only human visual consumption and not available in other ways, and display-centric information is being generated at such an increasingly faster rate that users themselves cannot keep up with the growing flow of information. Even if our eyes can momentarily capture the display output, our brains are often not able to keep up with all of it. To address this problem, computers must be enhanced in their own ability to process display-centric content to help users with all the information they see through their computers.

Screen readers have been developed to process display-centric content, especially for visually impaired users. They leverage the accessibility framework available on modern operating systems to provide some access to onscreen data. However, they are generally limited to watching changes to a foreground window only, or else they require the user to actively define hot spots, physical regions of the screen to scan for changes. Requiring blind and partially sighted users to specify these physical screen locations is at best cumbersome, and in many scenarios, it is difficult to select a cleanly differentiated region of the screen to watch. The end result is that only significantly reduced display-centric content is available to visually impaired users, and this gap in available information content grows only further as greater and richer
display information is available onscreen but not available via screen readers.

We introduce Capture, a novel display-centric text recorder that facilitates real-time access to both foreground and background onscreen text with low overhead. Capture provides an intelligent caching architecture that integrates with the accessibility framework available on modern operating systems. The caching architecture reduces the need to query the accessibility framework for onscreen data, which is important as these queries can impose high runtime overhead. Query results are cached so that additional requests for the same onscreen data are serviced by the cache instead of requiring additional queries. The cache is updated using a pull model in which multiple screen update notifications from the accessibility framework are coalesced so that they can be handled by a single query to the accessibility framework instead of multiple queries. This results in two key benefits. First, the caching architecture mitigates the high performance costs of using the accessibility framework while retaining its full functionality for accessing onscreen data. Onscreen data as well as the structure of the onscreen data is efficiently cached. Second, the caching architecture makes available not just the results of one query but a complete view of all onscreen data at a given time. All display-centric content is available for use in real time, including text and contextual data associated with both foreground and background windows.

Capture's unique combination of performance and functionality enables new ways of using onscreen content. The recorded data can benefit a variety of problem domains beyond that of DejaView, including assistive technologies, desktop search, auditing, and predictive graphical user interfaces. For example, screen readers can leverage Capture to provide access to display-centric content previously unavailable to visually impaired users. Capture does not just provide access to the latest changes to a foreground window, but instead can provide visually impaired users access to all of
the onscreen content. The information recorded by *Capture* could be used as a virtual hot spot, notifying the user of changes to specific desktop entities in a manner much easier than identifying physical screen locations. For example, users can be notified when specific applications and windows change, when GUI components change such as when a button appears on, when a status bar has reached 100% or has been halted part way for a period of time, or when a particular phrase appears anywhere on the desktop. Such a notification mechanism can enable blind or partially sighted users who rely on audio translation to more easily do out-of-order inspection of a document and avoid the need for periodic manual queries of physical screen locations. *Capture* can furthermore benefit such users to more easily get an overview of a foreign desktop, enabling them to better integrate in a workplace sharing computers with colleagues.

### 4.2 The Accessibility Framework

Assistive technologies help people with disabilities interact with computers by adapting the computer to the user’s abilities. For instance, screen readers convert onscreen displayed contents to speech, an essential aid for visually impaired users. Assistive technologies act as an intermediary between the application and the user. They provide special purpose forms of input and output, and monitor and control the program being run. To do so, assistive technologies must be able to access information from the applications. The accessibility framework is the infrastructure that connects assistive technologies and applications.

The relationship between assistive technologies, the accessibility framework, and desktop applications is illustrated in Figure 4.1. The accessibility framework acts as middleware between assistive technologies and applications. It provides a standard, consistent mechanism for applications and *observer programs* to exchange information,
and allows applications to expose rich information about the state of their GUI. Observer programs navigate and interrogate the collection of accessibility objects exposed by the applications. Observer programs also register to events, used to signal application state changes, in order to learn about and respond to GUI changes. For example, screen readers can be exposed to the type, name, location and current state of all GUI objects of an application, and be notified of any desktop event that leads to a user interface change.

The accessibility framework operates as a distributed system, in which applications and observer programs are independent entities that communicate through message passing. Observer programs and applications interact in a client-server manner. Applications play the role of the server, responding to requests for user interface elements, and sending event notifications to subscribed clients. Observer programs, such as accessibility aids, are the clients, requesting information from the applications to traverse the applications’ GUI components and extract their state.

In the accessibility framework, user interface elements on the screen are represented as a tree of accessibility objects, which usually closely mirrors the GUI tree.
structure. In this tree structure, the desktop is the root, application windows are immediate children, and elements within applications, including window tabs, frames, menus, buttons and others, are further descendants. Navigation between elements is mainly hierarchical: from parents to children and between siblings. Navigation between siblings is often logical in the context of ordered lists, such as menu items and window tabs. Contextual navigation is also possible, depending on the applications, such as between the description and the URL of a hyperlink.

Every accessibility object has a role attribute that describes the corresponding element in the user interface. Table 4.1 lists a subset of the roles defined in GNOME [1]. An object’s role provides context for the information exposed by the object. For instance, a tab list represents a series of page tabs, and its children are tab objects. Interaction with an object depends on its role. For example, a list object would report selected items, while a text object would simply report its contents. Some roles may be static, e.g., menu items that contain the same text indefinitely, while others, e.g., terminals, may update their state frequently.

Accessible objects additionally keep a set of properties and states. These provide specific information about the object, such as its name, screen position, text contents

<table>
<thead>
<tr>
<th>Role</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>An application instance</td>
</tr>
<tr>
<td>Frame</td>
<td>A top level window with a title bar, menu, etc.</td>
</tr>
<tr>
<td>List</td>
<td>A list of user-selectable objects</td>
</tr>
<tr>
<td>List item</td>
<td>An object that is an element of a list</td>
</tr>
<tr>
<td>Menu</td>
<td>An object found inside a menu bar</td>
</tr>
<tr>
<td>Menu item</td>
<td>A selectable action in a menu</td>
</tr>
<tr>
<td>Page tab list</td>
<td>A series of panels where one is selectable</td>
</tr>
<tr>
<td>Page tab</td>
<td>A selectable panel from a series</td>
</tr>
<tr>
<td>Terminal</td>
<td>An object that emulates a teletype or terminal</td>
</tr>
<tr>
<td>Text</td>
<td>A generic object that displays text</td>
</tr>
</tbody>
</table>

Table 4.1 – A subset of GNOME accessibility roles
Table 4.2 – A subset of GNOME accessibility properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>A string representing the object or its contents</td>
</tr>
<tr>
<td>Visible</td>
<td>Whether the object is visible</td>
</tr>
<tr>
<td>Showing</td>
<td>Whether the object and its ancestors are visible</td>
</tr>
<tr>
<td>Focused</td>
<td>Whether the object owns the focus</td>
</tr>
<tr>
<td>Iconified</td>
<td>Whether the frame has been shrunk or iconified</td>
</tr>
<tr>
<td>Selected</td>
<td>Whether the object is selected by the user</td>
</tr>
<tr>
<td>Text caret</td>
<td>Position of the text caret in the text</td>
</tr>
<tr>
<td>Attributes</td>
<td>A set of text attributes that apply to the object</td>
</tr>
<tr>
<td>Text</td>
<td>The text string of the object</td>
</tr>
<tr>
<td>Extents</td>
<td>The coordinates and geometry of the object</td>
</tr>
</tbody>
</table>

and text selections, and display state such as whether it is visible, in focus, or iconified. Table 4.2 lists a subset of accessibility properties in GNOME. The exact meaning of these may vary depending on the role of an object.

Changes to objects are communicated via event notification. Events are triggered for a variety of reasons. Examples include creation and removal of user interface elements, changes to text, cursor or mouse movement, text selection, state toggle between visible and invisible, focus shift between window tabs, and loading of a new document.

Table 4.3 lists a subset of accessibility events in GNOME. When the user interacts with the desktop, objects routinely join and leave the tree, while during idle times they remain mostly intact, save for occasional background activities. By and large, changes to a specific object do not modify the tree structure. For instance, typing text into an input box changes the state of the corresponding object only.

Figure 4.2 illustrates this representation for the GNOME text editor. It shows a screenshot of both the GUI window and a part of the corresponding accessibility tree (which was extracted with the Accerciser program in GNOME). In the latter, the left column gives the objects’ names, and indentation indicates parent-child relation-
<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>document:load-complete</td>
<td>Document has finished loading</td>
</tr>
<tr>
<td>document:reload</td>
<td>Document has finished reloading</td>
</tr>
<tr>
<td>document:attributes-changed</td>
<td>Document’s (text) attributes have changed</td>
</tr>
<tr>
<td>object:children-changed:add</td>
<td>A child object was added</td>
</tr>
<tr>
<td>object:children-changed:remove</td>
<td>A child object was removed</td>
</tr>
<tr>
<td>object:visible-data-changed</td>
<td>Data visible to the user has changed</td>
</tr>
<tr>
<td>object:visible-data-changed:remove</td>
<td>Data visible to the user has changed</td>
</tr>
<tr>
<td>object:text-changed:insert</td>
<td>One or more characters were inserted</td>
</tr>
<tr>
<td>object:text-changed:delete</td>
<td>One or more characters were deleted</td>
</tr>
<tr>
<td>object:text-selection-changed</td>
<td>Selected or highlighted text has changed</td>
</tr>
<tr>
<td>object:state-changed:defunct</td>
<td>Object was destroyed</td>
</tr>
<tr>
<td>object:state-changed:focus</td>
<td>Object has gained focus</td>
</tr>
<tr>
<td>object:state-changed:iconified</td>
<td>Object was minimized to the taskbar</td>
</tr>
<tr>
<td>state-changed:selected</td>
<td>Object was selected by the user</td>
</tr>
<tr>
<td>window:minimize</td>
<td>Frame was minimized to the taskbar</td>
</tr>
<tr>
<td>window:maximize</td>
<td>Frame was maximized to fill the screen</td>
</tr>
<tr>
<td>window:restore</td>
<td>Frame was restored from its minimized state</td>
</tr>
</tbody>
</table>

Table 4.3 – A subset of GNOME accessibility events

ships. The middle column indicates the role of the object, and the right column gives the number of children that an object has.

The accessibility framework is an integral part of modern desktops and is integrated into standard GUI toolkits [2, 114]. Standard widgets have default implementations provided by accessibility services in these toolkits. Custom widgets require that accessibility be specifically supported by the application. Applications without a GUI can also expose their state or text-based UI to make them accessible. Some applications, such as Firefox and OpenOffice, use their own accessibility implementations, which in addition to the user interface may also expose the structure of the documents browsed and displayed.
Chapter 4. Display-Centric Content Recording

(a) A screenshot of **GEdit**’s GUI window

(b) A screenshot of part of **GEdit**’s accessibility tree

**Figure 4.2** – GUI window of **GEdit** and part of the matching accessibility tree
4.3 Architecture

The primary aim of Capture is to provide display-centric text recording that facilitates real-time access to onscreen textual content. It must accomplish this with low overhead to not interfere with the user’s interactive experience, while simultaneously achieving accurate coverage of all onscreen data presented to the user. Capture should also provide this functionality transparently without any modifications to applications, window systems, or operating system kernels. To achieve these goals, Capture builds on the standard accessibility framework available on modern operating systems, which provides the ability to retrieve textual and contextual information from the display.

The desktop accessibility framework is paramount to the transparency of Capture, but suffers from an important limitation. Because it is a distributed architecture based on message passing, the accessibility framework is sensitive to communication latencies between processes. A distributed model is particularly useful since it lets applications control what GUI state they expose, and allows any program to query any applications in an efficient manner. However, it also means that intense querying of multiple applications through the accessibility framework, as is needed for display-centric text capturing, can become a bottleneck due to query latencies.

While enabling many programs to query applications arbitrarily is efficient in a distributed model, in reality it is an uncommon use-case: only rarely more than a single observer program is running on a desktop. Rather, a typical scenario includes one or a few observer programs that repeatedly query one or more applications, as do screen readers and automatic GUI testing tools [70]. Capture takes this a step further by gathering display-centric information across all applications. The naive approach of using queries in Capture in a similar manner as used by traditional screen readers
is problematic because of significant differences in the *scope* of queries and the *type* of queries required.

In terms of scope, screen readers are window-centric. They are only interested in, and post queries to, the current application in focus. Even within a single application, more often than not, screen readers narrow their attention to a specific window and specific frame that correspond to the GUI element currently in focus. In contrast, *Capture* is desktop-centric, and must track multiple applications as it seeks to record information of the desktop as a whole. Additionally, it considers all GUI components in each application rather than the specific one in focus.

In terms of query types, a single query is limited in its capabilities: it may only retrieve a single piece of information from a single GUI object. Screen readers pursue textual information, and therefore typically use a single query to retrieve the text from the current GUI object in focus. Screen readers also rely on event payloads to learn about changes to that text. In contrast, *Capture* is also concerned about contextual information to support a broader range of applications, which requires multiple queries to be retrieved in full.

Capturing the onscreen text of an entire desktop involves extracting significantly more information from the accessibility framework than traditional screen readers do. Screen readers typically tolerate query latencies because they target only the GUI object in focus and issue few queries. They need not perform faster than at interactive rates, and are usually bounded by the ability of visually impaired users to absorb audible data, which rarely exceeds 600 words per minute [77]. Conversely, retrieving the full accessibility data of a whole desktop comprising of multiple applications, each with hundreds to thousands of accessibility nodes, could generate a multitude of queries. With typical latencies in the order of a few milliseconds per query, this would become prohibitively expensive. A typical scan does not only ex-
hibit unacceptable response times, but also generates an excessive load on the system and produces inaccurate and outdated results as accessibility data may change in the interim. Furthermore, because queries are synchronous, issuing a large volume of queries can impair interactivity, while issuing too few queries may result in missed data if the screen is changing quickly.

4.3.1 Overview

In addressing this challenge, it is important to understand that the current distributed model of the accessibility framework is ill-suited for this kind of usage patterns, suggesting the use of a different model. Building on this observation, Capture works to transform the existing message passing model to a centralized model, through which observer programs can efficiently and continuously access the entire onscreen contents at any time without excessive querying. Capture integrates with the accessibility framework, and does not alter how applications interact with the accessibility framework. In this way, Capture preserves the benefits of the distributed model, namely applications’ independence in exposing the GUI state, while providing an effective means to access the information.

To accomplish this, Capture introduces a novel caching architecture that integrates with the accessibility framework to reduce the need to query the accessibility framework for onscreen data. Query results are cached so that additional requests for the same onscreen data are serviced by the cache instead of requiring additional queries. The cache is updated using a pull model in which multiple event notifications from the accessibility framework are coalesced so that they can be handled by a single query to the accessibility framework instead of multiple queries. The caching architecture has two key benefits. First, it mitigates the high cost of using the accessibility frame-
work while retaining its full functionality for accessing onscreen data. Onscreen data as well as the structure of the onscreen data is efficiently cached. Second, it makes available not just the results of one query but a complete view of all onscreen data at a given time. All display-centric content is available for use in real-time, including data and metadata associated with both foreground and background windows.

Figure 4.3 depicts Capture’s caching architecture and its integration with the accessibility framework. At the core of the architecture are two components: a mirror accessibility tree and an event handler. The mirror tree (§ 4.3.2) serves as the cache that mirrors the onscreen textual and contextual information. It is updated in response to changes to onscreen data by the event handler. The event handler (§ 4.3.3) continuously selectively listens to accessibility event notifications, and responds by querying about display changes and updating the local mirror tree. The two components together provide an efficient cache that accurately reflects the entire onscreen state at any time. The front-end of Capture consists of one or more output modules (§ 4.3.4) that can efficiently access the onscreen information available in the cache, to perform custom tasks, such as content logging.
4.3.2 Mirror Tree

To initialize the mirror tree at startup, *Capture* populates it with data scanned from the initial desktop accessibility tree. Textual and contextual information is the primary data cached in the mirror tree, but nodes also store information beyond the queried accessibility data, which is useful to improve navigation and management of the tree. Examples include a pointer in each node to quickly locate the parent application of the corresponding GUI elements, and timestamps that indicate when data was last updated.

Nodes are added to the mirror tree when new elements appear onscreen. New nodes are discovered in two ways: first, explicitly by an event that refers to a node previously unknown; second, implicitly through scanning the children of an element that corresponds to a new node, or rescanning that of an existing one in response to a state update. To attach the new node at the proper position in the mirror tree, *Capture* first queries for the parent element in the accessibility tree. If the corresponding parent node is not found in the mirror, it means that the parent node itself is new, and *Capture* must add that parent before it can proceed with the current node. This may continue up the ancestry chain until a known node is eventually found (and at least one such node exists, namely, the root of the desktop GUI hierarchy). Once the new node is attached, its data is retrieved via accessibility queries. If the element has descendants in the GUI tree, *Capture* iterates through its children and recursively adds them to the tree.

Nodes are removed from the mirror tree when the corresponding elements are gone from the screen. Deleted nodes are discovered in three ways: first, explicitly by events that indicate deletion of nodes, such as “child-changed:deleted” for node removal from a parent and “object:state-changed:defunct” when a node becomes defunct; second,
implicitly when a defunct state is found for a node during a standard query on it; third, indirectly when the node belongs to a subtree that has been removed. Because freeing a deleted node incurs overhead on the accessibility framework, node removals are preferably deferred to when the system is otherwise idle.

### 4.3.3 Event Handler

The rate of events and the processing needed to handle them both vary considerably depending on the user’s behavior. First, desktop usage is of a bursty nature, and user activity may trigger a barrage of accessibility events followed by idle times. Second, which events are generated depends on the user’s activity and the applications in use. For example, typing input data produces “text-changed” events for each keystroke, while loading a web page in a browser produces a series of “child-changed” events as the accessibility tree for the web page is constructed, followed by a “load-complete” event. Third, the amount of work involved to process an event depends on its type. For example, for a “text-changed” event, a single query about the object’s new text suffices to update the data in the mirror tree. However, events that are indicative to changes to the accessibility tree structure, such as “child-change” after adding an object, typically involve multiple queries about the new object, and exploring its descendants (if any) recursively.

Because of query latencies, the event handler need not only listen to events and act on them, but it must also do so in an intelligent way to reduce the runtime overhead by skipping unnecessary queries, and without sacrificing accuracy or coverage of the onscreen content. To accomplish this, the event handler employs five mechanisms to boost event processing: enqueuing and dequeuing, grouping, deferring, reordering, and filtering and dropping of events. These mechanisms aim to avoid handling of
redundant events, which improves performance and prioritizes high-value information for better coverage.

First, since events usually arrive faster than they can be processed, Capture maintains a queue of backlogged events. Incoming events that are not tended to promptly are appended to the queue for later processing. The event handler continuously de-queues events from the head of the queue and processes them.

Second, Capture groups multiple events that have a common source object in a single entry in the queue. To do so, event entries in the queue are indexed by their source object and include a bitmap of event types. In the bitmap, each event type has a predefined bit position. When an event arrives from an object, Capture looks up an entry for the object in the event queue. If an entry is not already found in the queue, a new entry will be enqueued and the bit corresponding to the event type marked. Conversely, if an entry is found, as in the case, for example, of subsequent events from the same object, Capture will set the bit corresponding to the type of the new event in that entry, and so on.

Grouping together events from the same source has two benefits. First, multiple instances of each event type are folded together, so that Capture only processes one event of each type for each object and discards the rest. This is useful in situations where a GUI element fires multiple events of the same type. For instance, consider a status bar for messages, where new messages replace older ones, producing “text-changed” events. In this case it is unnecessary to issue multiple queries for deferred events, because all queries will return the data related to the last event, which is the current onscreen data. Similarly, a terminal application also produces “text-changed” events with keystrokes, but now data accumulates. Here, too, one query that returns the most current data suffices for all deferred events. This data, unlike before, will be an aggregate of the intermediate states.
Another benefit of grouping events is that it can exploit overlaps in semantics of events of different types to eliminate redundant work when processing events. For example, consider an object that produces a “child-changed:delete” event followed by a “child-changed:added” event. Processing each event involves iterating over all of an object’s children using accessibility queries. But if merged, both events can be processed in a single iteration. Similarly, consider a user editing text, producing multiple instances of both “text-changed:inserted” and “text-changed:deleted” events. Capture handles both with a query about the object’s new text, and a single query suffices for both.

Events sometimes carry useful information related to them, such as what character was typed or what text was inserted. This information is particularly important if, by the time an event is finally processed, the corresponding state was lost because the object’s content has changed further. For this reason, Capture keeps the payload of individual events after merging them together.

A mechanism related to grouping is deferring of events, used during bursty periods of activity that generate a surge of events all related to a single GUI “transaction”. For example, when downloading a page, Firefox exposes the HTML components in its accessibility tree, producing a spree of events. These should be viewed as part of a single atomic transformation of the user interface, rather than individual changes. When Capture identifies a burst of events due to such a transaction, it defers the handling of some (or all) events altogether until the transaction completes, to avoid the undesired application slowdown while the transaction is in progress, and to avoid reporting intermediate state that was not meant to be displayed.

A third mechanism to improve event handling is reordering of events in the queue. Reordering aims to achieve better accuracy and coverage by classifying events as either high or low priority, and giving faster service time to events of higher priority.
Priorities are assigned in accordance with the importance of the event. For example, while web page loading involves many “child-changed” events, it is the final “load-complete” event that marks when the page is ready. Thus, it is desirable to handle the latter as soon as it arrives, and before other potentially already queued events. Reordering provides a means to execute this policy.

Fourth, to further avoid unnecessary processing, Capture filters uninteresting events, and selectively discards stale events that no longer convey useful information. Filtering of uninteresting events is accomplished by simply not registering for them in the first place. Examples include events that do not reflect changes to the onscreen content, such as “text-caret-moved” and “mouse:button:...”. Unlike the filtering of dull events, selecting which events to discard is more challenging, and requires a way to detect when a particular event becomes stale. In our context, an event becomes stale if the object to which it refers has been updated after the arrival of the event.

To be able to efficiently identify events that become stale, Capture keeps track of the relative order of event arrivals and of updates to nodes in the mirror tree. Specifically, it maintains timestamps not only on mirror tree nodes, but also on events queued for processing. Timestamps of tree nodes are updated when their data is queried, and timestamps of queued events are updated when new events (for the corresponding object) arrive. The event handler compares the timestamps of a dequeued event against that of the corresponding node. Events with older timestamps are aged (marked stale) and skipped.

The main benefit of this approach is the ability to ignore events that do not carry useful information, but which are not subject to grouping. The two mechanisms, grouping and aging, complement each other: grouping affects cases where events share a common source, while aging affects such cases where an event from the same source arrives when a previous one is being processed, or when events from different sources
affect the same node. Continuing the example of loading a web page, processing the “load-complete” event will assign a newer timestamp to the entire subtree that describes the web page, and render most of the preceding “child-changed” events stale.

4.3.4 Output Modules

One or more output modules can be plugged in to Capture to use the collected information. They enjoy efficient access to the onscreen information that is available in the cache, and can register to be notified when the state of the cache changes due to modifications of onscreen content. Output modules can perform a range of useful tasks. Continuous or periodic logging of the cache state, advanced assistive technology services for visually impaired users, and a GUI extension that reacts to certain text patterns onscreen are just a few examples.

Output modules execute in a separate process to isolate them from the main Capture application. They interact with Capture using two communication channels: data channel and control channel. The data channel provides direct access to the cache through shared memory. In particular, Capture allows output modules to map a read-only copy of the cache data structures. Using shared memory has two performance benefits. First, it is the fastest form of data transfer. Second, it enables direct and asynchronous random access to the mirror tree so that output modules can traverse the data independently of Capture. The control channel is used for interaction between output modules and Capture in a client-server style. Output modules use the channel to register for notifications about changes to the cached data, and to request to lock their read-only copy of the cache while they traverse the data to protect its integrity. Capture uses the channel to deliver event notifications.
4.3.5 Limitations

Capture has two main limitations as a generic display-centric text recorder. First, in relying on the accessibility framework it depends on the correctness and completeness of applications’ accessibility support. Capture can not derive text from the (increasingly rare) applications for which the accessibility interfaces are either non-existing or unreliable. 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 Capture to extract textual information from them.

Second, due to the accessibility framework’s event-driven operation, Capture’s coverage accuracy highly depends on the response time to notification. In the lag between when a change in an application’s GUI occurs and when Capture queries the application about it, the state may evolve further to the point that the original change is gone. In some cases the changes can be additive, e.g., when a user types text while working with a word processor, such that no loss of data will occur due to delayed queries.

There are three factors that affect the overall response time. First, the application logic may decide to aggregate multiple changes before triggering an event. For instance, a word processor may report the outcome of multiple keystrokes as a single accessibility update. Another factor is Capture’s processing needed to handle events, which depends on the number of pending events and their types. For example, high priority events delay the handling of lower priority events. The third factor is the time needed to deliver messages between the application and Capture (and if the sys-
tem is loaded, the time until being scheduled to run). Although fast rate of changes can adversely affect the coverage of *Capture*, our experimental results presented in the following section demonstrate that *Capture* is fast enough to accurately capture display-centric data on a range of applications during normal interactive use.

### 4.4 Evaluation

We have implemented *Capture* for Linux and the GNOME desktop with the GNOME Accessibility Toolkit. Using this unoptimized prototype, we measured the effectiveness of *Capture* in recording onscreen data for real desktop applications in comparison to using the widely used Orca screen reader [123] for Linux. To provide a conservative comparison with Orca, we disabled text-to-speech translation to minimize its recording overhead, and we only measured recording of text as opposed to recording of additional metadata which is done by *Capture* but not Orca. We measure effectiveness both in terms of text coverage and runtime performance overhead. For text coverage, we measure the percentage of onscreen data recorded in terms of number of correct text characters recorded versus the number of onscreen text characters generated for each application workload. For runtime overhead, we measure the normalized runtime of the application workloads compared to running without any recording or using any accessibility framework.

All experiments were run using the same hardware and software. For hardware, we used a Dell Vostro 200 desktop with Intel Pentium E2140 dual-core 1.60 GHz CPU, 4.0 GB RAM, and a 500 GB local disk. For software, we used Ubuntu 10.04 LTS with a Linux 2.6.32 kernel. To interface with the accessibility framework, we used AT-SPI accessibility libraries *libatk* and *libatspi* version 1.30, *libbonobo* version 2.24.32, and *liborbit2* version 2.14.18.
Chapter 4. Display-Centric Content Recording 68

Table 4.4 – Application scenarios

<table>
<thead>
<tr>
<th>Application</th>
<th>Benchmark description</th>
<th>Exposed text</th>
<th>Textual nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe Reader 9.3.2</td>
<td>Open 1 page accessibility tagged 220 KB PDF file, then close it</td>
<td>entire</td>
<td>multiple</td>
</tr>
<tr>
<td>PDF reader</td>
<td></td>
<td>document</td>
<td>nodes</td>
</tr>
<tr>
<td>Firefox 3.6.3</td>
<td>Download 56 web pages in sequence from a web server, 228 KB visible text in total</td>
<td>entire</td>
<td>multiple</td>
</tr>
<tr>
<td>web browser</td>
<td></td>
<td>document</td>
<td>nodes</td>
</tr>
<tr>
<td>GEdit 2.30.2</td>
<td>Type total of 500 characters split into 80 characters per line</td>
<td>entire</td>
<td>single</td>
</tr>
<tr>
<td>text editor</td>
<td></td>
<td>document</td>
<td>node</td>
</tr>
<tr>
<td>GNOME-Terminal 2.29.6</td>
<td>more page-by-page display of 4769 lines, 1534 pages (2.6 MB in total)</td>
<td>visible</td>
<td>single</td>
</tr>
<tr>
<td>terminal emulator</td>
<td></td>
<td>text only</td>
<td>node</td>
</tr>
<tr>
<td>OpenOffice Impress 3.2.0</td>
<td>Page through 10 slides, each containing 1024 characters of content</td>
<td>visible</td>
<td>multiple</td>
</tr>
<tr>
<td>spreadsheet</td>
<td></td>
<td>text only</td>
<td>nodes</td>
</tr>
<tr>
<td>OpenOffice Writer 3.2.0</td>
<td>Open and scroll through a 10-page text document page by page</td>
<td>visible</td>
<td>multiple</td>
</tr>
<tr>
<td>word processor</td>
<td></td>
<td>text only</td>
<td>nodes</td>
</tr>
<tr>
<td>Pidgin 2.6.6</td>
<td>Send 10 1024-character messages using the AIM protocol</td>
<td>entire</td>
<td>single</td>
</tr>
<tr>
<td>IM client</td>
<td></td>
<td>document</td>
<td>node</td>
</tr>
<tr>
<td>Thunderbird 3.0.4</td>
<td>Read 10 emails, each 800 characters and entirely visible onscreen</td>
<td>entire</td>
<td>single</td>
</tr>
<tr>
<td>email client</td>
<td></td>
<td>document</td>
<td>node</td>
</tr>
</tbody>
</table>

We used the applications workloads listed in Table 4.4, which represent various common desktop usage cases. Except for Firefox, which used the iBench 1.5 web page benchmark, all other workloads used documents whose pages fit entirely visible onscreen. Our measure of text coverage is based only on the text visible onscreen and does not include whether the system successfully recorded data that was not visible onscreen.

Since applications decide themselves how to expose data through the accessibility framework, they deliver different information regarding their textual context. Applications differ in what information they deliver about documents that do not fit entirely in the displayed window, such as a PDF document with multiple pages or a long web page. Some of the applications provide information on the entire document at once, including pages not yet visible onscreen, while other applications provide information only for the page visible onscreen at a given time. Applications also differ in whether they expose all of the textual information in a single tree node in the ac-
cessibility framework, or use multiple tree nodes. These two properties are indicated for each workload by the “Exposed text” and “Textual nodes” columns in Table 4.4.

Since real interactive desktop usage typically involves some amount of user think-time after receiving certain display output and before sending additional input, each application workload includes such think-time by delaying the start of the next input some time after the end of the previous output. We used a range of think-times for each workload between 50 ms and 1000 ms. The shorter times are conservative values that were much faster than typical human usage to stress test the system, while the longer ones provide a more realistic scenario that matches common behavior.

Think-times were integrated into workloads as follows: for the Adobe Reader PDF viewer before closing the document; for the Firefox web browser before displaying the next web page; for the GEdit text editor before typing the next character; for the GNOME Terminal terminal emulator before displaying the next page; for the OpenOffice.org Impress presentation software before displaying the next slide; for the OpenOffice.org Writer word processor before scrolling down to the next page; for the Pidgin instant messaging client before sending the next message; for the Thunderbird mail client before reading the next email message.

To ensure the repeatability of our experiments, we used the Linux Desktop Testing Project [103] (LDTP) scripting environment to simulate and automate virtual user interaction and to control think-time. We used LDTP to inject input in the form of keystrokes and mouse clicks to applications at a preset rate, and then measured how much of the expected output was actually captured. To simplify the execution of repeatable experiments, we used VNC version 4.1.1 to run each desktop session from scratch, with display resolution of 1024x768. The VNC client was disconnected during performance measurements to minimize any variability from different display speeds due to any network interference.
Chapter 4. Display-Centric Content Recording

Table 4.5 – Summary of benchmark results with think-time of 1000 ms

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native time</th>
<th>Capture</th>
<th>Orca</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Norm. time</td>
<td>Coverage</td>
</tr>
<tr>
<td>acroread</td>
<td>58 s</td>
<td>1.32</td>
<td>100%</td>
</tr>
<tr>
<td>firefox</td>
<td>71 s</td>
<td>1.03</td>
<td>100%</td>
</tr>
<tr>
<td>gedit</td>
<td>80 s</td>
<td>1.01</td>
<td>100%</td>
</tr>
<tr>
<td>terminal</td>
<td>60 s</td>
<td>1.00</td>
<td>100%</td>
</tr>
<tr>
<td>ooimpress</td>
<td>52 s</td>
<td>1.00</td>
<td>100%</td>
</tr>
<tr>
<td>oowriter</td>
<td>59 s</td>
<td>1.00</td>
<td>100%</td>
</tr>
<tr>
<td>pidgin</td>
<td>41 s</td>
<td>1.03</td>
<td>100%</td>
</tr>
<tr>
<td>thunderbird</td>
<td>62 s</td>
<td>1.00</td>
<td>100%</td>
</tr>
</tbody>
</table>

4.4.1 Performance Overhead

Table 4.5 shows the runtime overhead of recording the application workloads using Capture versus Orca, for a think-time value of 1000 ms, except for GEdit for which the think-time was 100 ms between characters. The “Native time” column shows the completion time for each workload for native execution without recording. The “Norm. time” columns show the respective normalized runtime performance overhead when using Capture and Orca. The “CPU Load” columns show the average processor load (as reported by the top utility) during the execution of the workloads.

In general, the execution overhead was quite low for both Capture and Orca, under 3% for all workloads. Capture’s performance was comparable to Orca’s in terms of execution overhead, and somewhat better in terms of overall processor load. Our results show that that Capture incurs only modest performance overhead even for display-intensive workloads, and in general does not incur more overhead than screen readers such as Orca despite delivering significantly more display content.
4.4.2 Single Application Text Coverage

Table 4.5 also shows the text coverage achieved when recording the application workloads using Capture versus Orca. 100% coverage means that all onscreen text data displayed throughout the application workload was recorded successfully. The results show that Capture achieved 100% coverage for all of the application workloads.

In contrast, Orca was only able to achieve 100% coverage for two of the application workloads and instead delivered fairly low coverage for a majority of the workloads. This is despite the fact that the application workloads used provide the most favorable comparison with Orca since only a single foreground application workload is being run in these experiments. Orca’s coverage is under 60% for half of the application workloads. For some applications, such as Adobe Reader and OpenOffice Impress, Orca fails to record any of the onscreen text. These results show the benefits of Capture’s caching architecture in being able to combine complete recording of onscreen text while minimizing recording overhead.

Figure 4.4 and Figure 4.5 show the coverage of Capture and Orca, respectively, with varying think-times of 50 ms, 200 ms, 400 ms, 700 ms, and 1000 ms. These results show that the gap in coverage performance between Capture and Orca increases as the think-time shortens. While both systems display lower coverage with the decrease in think-time, the negative impact is significantly stronger on Orca, which lacks the ability to cope efficiently with the intense flux of accessibility events. Because these workloads were designed to stress the onscreen text capture performance, the coverage results of Capture are much worse than what would be perceived by real users. For example, the 100 ms delay used for GEdit would translate to a typist who could enter 1200 words per minute, an order of magnitude faster than what is possible for a very fast human typist.
Figure 4.4 – Capture coverage vs. think-time

Figure 4.5 – Orca coverage vs. think-time
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Capture coverage</th>
<th>Orca coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe Reader</td>
<td>53%</td>
<td>0%</td>
</tr>
<tr>
<td>Firefox</td>
<td>79%</td>
<td>1%</td>
</tr>
<tr>
<td>GEdit</td>
<td>93%</td>
<td>33%</td>
</tr>
<tr>
<td>GNOME-Terminal</td>
<td>95%</td>
<td>31%</td>
</tr>
<tr>
<td>OO-Impress</td>
<td>93%</td>
<td>0%</td>
</tr>
<tr>
<td>OO-Writer</td>
<td>80%</td>
<td>17%</td>
</tr>
<tr>
<td>Pidgin</td>
<td>82%</td>
<td>33%</td>
</tr>
<tr>
<td>Thunderbird</td>
<td>90%</td>
<td>3%</td>
</tr>
</tbody>
</table>

Table 4.6 – Multiple-application text coverage

4.4.3 Multiple-Application Text Coverage

Table 4.6 shows the text coverage achieved when recording the application workloads in the foreground window while also running additional application workloads in the background. The background application workloads used in all cases were an additional instance of the Firefox and GNOME Terminal workloads. We measure the text coverage across all of the running applications, including both foreground and background windows, using Capture versus Orca.

The results show that Capture achieves nearly 100% coverage for most of the application workloads. In contrast, Orca’s coverage is well under 50% for all application workloads, and is close to 10% on most of them. because Orca only tracked the text from the foreground application and ignored the background applications. Orca achieved higher coverage for Impress than before, because it did record some text from the Firefox application that ran in the background. These results show that Orca’s ability to record onscreen text is largely limited to foreground applications, and further impaired in the presence of accessibility events due to background applications. The results highlight the benefits of Capture’s caching architecture in being able to achieve complete recording of onscreen text efficiently.
4.4.4 Tree Characteristics

We collected statistics about the distribution of accessibility component types in different applications, and their memory usage. This data is useful to better understand the how applications interact with the accessibility framework. For instance, applications tend to use a larger number of nodes that hold text than not, meaning that Capture is likely to spend substantial time in querying about the text of descendants of a new node.

Figure 4.6 and Figure 4.7 show the count of accessibility nodes in the mirror tree and their memory usage, respectively. The data is broken down into nodes containing *no text*, menu or button *widget* with text, and nodes containing *general* text (other than menu and button text). We distinguish between menus and buttons that are typically static from general text which tends to change more often. To examine how the distribution changes with input data, we repeated the measurements twice: a *base* case without data, followed by a *doc* case with the input data or document.

The count of nodes reflects differences in how applications expose their GUI state through the accessibility framework. For Adobe Reader, for instance, the differences in node counts between the base and the document cases indicates that Adobe Reader splits the document into multiple accessibility nodes. For this particular 169 KB document, the tree grew by nearly 800 nodes, an increase of 200% compared with the base case. The difference for other applications is smaller, often because they expose the entire content of each document in a single node, as do GEdit and GNOME Terminal. For instance, GEdit’s tree maintains the same size of approximately 450 nodes even after opening a 106 KB document.

The OpenOffice.org Impress scenario is an anomaly in that when a regular document is open, it reveals fewer accessibility nodes than a blank document. This
Figure 4.6 – Mirror tree size

Figure 4.7 – Mirror tree memory
discrepancy can be attributed to the presentation template selection dialog that appears beside a blank presentation; the dialog reveals its contents via a number of accessibility nodes.

To better understand how Capture’s performance depends on the structure (or lack of structure) of documents, we investigated two sets of web pages in Firefox: documents in one set had a single node of increasing sizes, and documents in the other had an increasing number of nodes. We measured the amount of time that Capture takes to scan the entire mirror tree on each of the documents.

Figure 4.8 and Figure 4.9 show the results for the series varying the node size and the number of nodes, respectively. The time to capture an HTML document was insensitive to the amount of the text contained in a single node, whereas varying the number of nodes on the page caused an increase in runtime. The time to build the tree ranged from under 4 ms per node on average when there was one node of document text, to 15 ms per node on average when the document had 10,000 nodes. These results demonstrate a superlinear increase in the query time per node. The growing number of nodes implies more queries in a small span of time, which causes congestion in the application (in this example, Firefox) trying to service the many
requests. In contrast, the results demonstrate the relatively small change in runtime when increasing the size of text contained in a single node, suggesting that data bandwidth is not a significant performance issue for accessibility queries.

4.5 Lessons Learned

In designing Capture, we have addressed a number of challenges related to the accessibility framework. One key difficulty emanates from the performance limitation of a client-server model, namely, long turnaround time for queries to applications. Our results show that the dominant factor affecting performance is the number of queries rather than the amount of data transferred in each query. In this section we offer design principles to help avoid the types of pitfalls that we encountered. Specifically, we present four design principles distilled out of our experience that aim to extend the accessibility framework interfaces for efficient and effective use in the context of our (and future) work.

- **Bulk Data Transfer.** Empower callers to request information about multiple accessible objects in a single query. Using a single bulk transfer request to replace otherwise numerous normal requests will dramatically reduce the aggregate cost of accessibility requests, and mitigate the performance bottleneck and the problems related to it. Bulk transfer can come in two flavors. One is a “vector query” that combines multiple regular queries into a single request, e.g., to collect the text of multiple objects at once. The other is a “recursive query” that extends to the entire subtree of a given accessibility object, e.g., for fast acquisition of an entire application’s subtree.
• **Views and Filters.** Allow callers to obtain a focused view of the data efficiently. For instance, a caller may request a list of all accessibility objects corresponding to user interface elements that have non-trivial contents, or that are currently visible in a given window. Such an interface would be a perfect match for applications that deal with huge accessibility hierarchies, like *OoCalc*.

• **Burst Event Delivery:** Allow an application to raise a horde of events efficiently, and allow callers to collect information about a batch of events using a single accessibility request. This can be additionally refined to allow “hierarchical events” that allude to an entire hierarchy, such as a single event to indicate the deletion of an entire subtree instead of a series of events to delete each and every node in it.

• **Transaction Boundaries.** Allow an application to disclose the start and end of a series of related events that can be viewed as a coherent transaction that transforms the user interface. A caller may choose to defer the handling of some (or all) events until the transaction completes, and avoid the undesired application slowdown while the transaction is in progress. A precise indication of when a transformation completes is also useful to be able to associate a precise timestamp to mirror tree nodes as they are updated.

In our experience, different applications vary widely in their interpretation of the accessibility framework, and it is very difficult to design a generic, application-agnostic approach that fits all scenarios and performs well. Rather, an application-aware logic is mandatory to accurately infer important features of the accessibility tree. This discrepancy stems mostly from loose definitions of the accessibility interfaces; We believe that better accessibility guidelines combined with better awareness of application developers would significantly reduce the extent of this issue.
4.6 Summary

In this chapter we presented DejaView’s display-centric content recording for extracting text from the display so that displayed content can be indexed and later searched. We introduced Capture, a novel display-centric text recorder that facilitates real-time access to onscreen textual and contextual information from both foreground and background windows. Capture provides an intelligent caching architecture that integrates with the standard accessibility framework available on modern operating systems to continuously track onscreen text and metadata, without modifying existing system components. Our experimental results show that Capture imposes only modest performance overhead on a wide range of desktop applications while being able to accurately record all onscreen textual data even when applications are generating updates to onscreen data at a high frequency. Compared to a screen reader, Capture records a substantially higher percentage of onscreen text for both foreground and background applications, and is able to record common application workloads that the screen reader does not record at all.
Chapter 5

Virtual Execution Environment

In this chapter, we present DejaView’s virtual execution environment, which plays a crucial role in supporting the ability to record and later revive live execution state. This virtual execution environment decouples the user’s desktop computing environment from the underlying operating system, so that DejaView can continuously checkpoint and later revive entire live desktop sessions. Furthermore, it confines the user’s desktop session in a private environment so that multiple live desktop sessions can run concurrently in isolation from one another. DejaView’s virtual execution environment transparently encapsulates a user’s desktop computing session in a private virtual namespace. By providing a virtual namespace, revived sessions can use the same operating system resource names as used before being checkpointed, even if they are mapped to different underlying operating system resources upon revival. By providing a private namespace, revived sessions from different points in time can run concurrently and use the same operating system resource names inside their respective namespaces, yet not conflict among each other.

DejaView leverages the standard interface between applications and the operating system (i.e., operating system virtualization) to provide its virtual execution environ-
ment in a manner that is transparent to both applications and the operating system kernel. To the applications running within, the virtual execution environment looks like a regular system, provides the same interfaces, and thereby does not require applications to be adapted. By interposing between applications and the underlying operating system kernel, the operating system kernel does not need to be modified either. Furthermore, operating system virtualization is a lightweight mechanism that imposes low overhead as it operates above the operating system instance to encapsulate only the user’s desktop computing session, as opposed to an entire machine instance.

We present a detailed discussion of key implementation issues and challenges in providing operating system virtualization in a commodity operating system. We compare alternatives for implementing operating system virtualization at user-level vs. kernel-level, discuss performance costs for methods of storing virtualization state, and examine subtle race conditions that can arise in implementing operating system virtualization. Some operating systems are gradually incorporating virtualization support by making pervasive changes to the operating system kernel [106]. We describe an approach of implementing operating system virtualization in a minimally invasive manner by treating the operating system kernel as an unmodified black box, demonstrating how commodity operating systems can embrace operating system virtualization with minimal changes. Using this approach, we have implemented a Linux operating system virtualization prototype entirely in a loadable kernel module, and present quantitative results demonstrating that such a minimally invasive approach can be done with very low overhead.
5.1 Operating System Virtualization

Virtualization essentially introduces a level of indirection to a system to decouple applications from the underlying host system. This decoupling can be leveraged to provide important properties such as isolation and mobility, providing a myriad of useful benefits. These benefits include supporting server consolidation by isolating applications from one another while sharing the same machine, improved system security by isolating vulnerable applications from other mission critical applications running on the same machine, fault resilience by migrating applications off of faulty hosts, dynamic load balancing by migrating applications to less loaded hosts, and improved service availability and administration by migrating applications before host maintenance so that they can continue to run with minimal downtime.

While virtualization can be performed at a number of different levels of abstraction, providing virtualization at the correct level to transparently support unmodified applications is crucial in practice to enable deployment and widespread use. The two main approaches for providing application transparent virtualization are hardware virtualization and operating system virtualization. Hardware virtualization techniques [47, 147, 169, 180] virtualize the underlying hardware architecture using a virtual machine monitor to decouple the operating system from the hardware so that an entire operating system environment and associated applications can be executed in a virtualized environment. Operating system virtualization techniques [21, 122, 124, 138, 148, 173, 181] virtualize the operating system to decouple applications from the operating system so that individual applications can be executed in virtualized environments. Hardware virtualization and operating system virtualization techniques each provide their own benefits and can provide complementary functionality.
Operating system virtualization provides a fine granularity of control at the level of individual processes or applications, which offers several benefits over the hardware virtualization abstraction that works with entire operating system instances. For example, operating system virtualization can enable transparent migration of individual applications, not just migration of entire operating system instances. This finer-granularity migration provides greater flexibility and results in lower overhead [95, 124]. Furthermore, if the operating system requires maintenance, operating system virtualization can be used to migrate the critical applications to another running operating system instance. By decoupling applications from the operating system instance, operating system virtualization enables the underlying operating system to be patched and updated in a timely manner with minimal impact on the availability of application services [135]. Hardware virtualization alone cannot provide this functionality since it ties applications to an operating system instance, and commodity operating systems inevitably incur downtime due to necessary maintenance and security updates.

Operating system virtualization isolates processes within a virtual execution environment by monitoring their interaction with the underlying operating system instance. Similar to hardware virtualization [131], applications that run within the virtual environment should exhibit an effect identical to that demonstrated as if they had been run on the unvirtualized system. In addition, a statistically dominant subset of the interaction of applications with system resources should be direct to minimize overhead.

We classify operating system virtualization approaches along two dimensions, host-independence and completeness. Host-dependent virtualization only isolates processes while host-independent virtualization also decouples them. The distinction is that host-dependent virtualization simply blocks or filters out the namespace between
processes, while host-independent virtualization provides a private virtual namespace for the applications’ referenced operating system resources. The former does not support transparent application migration since the lack of resource translation tables mandates that the resource identifiers of an application remain static across hosts for a migrating process, which can lead to identifier conflicts when migrating between hosts. Examples of host-dependent virtualization include Linux V Servers [173] and Solaris Zones [138]. Host-independent virtualization encapsulates processes in a private namespace that translates resource identifiers from any host to the private identifiers expected by the migrating application. Examples of this approach include Zap [95, 124] and Capsules [148]. We refer to this virtual private namespace as a pod, based on the terminology used in Zap.

In terms of completeness, partial virtualization virtualizes only a subset of operating system resources. The most common example of this is virtual memory, which provides each process with its own private memory namespace but doesn’t virtualize any other operating system resources. As another example, the FreeBSD Jail [87] abstraction provides partial virtualization by restricting access to the file system, network, and processes outside of the jail, but does not regulate SysV interprocess communication (IPC) mechanisms. While partial virtualization has been used to support tighter models of security by limiting the scope of faulty or malicious processes, it can be unsafe if direct or indirect paths exist through which processes inside the environment can access resources outside or even break out of the environment. The chroot environment in Unix is a notorious example of a file system partial virtualization mechanism that has serious security shortcomings [33].

Complete virtualization virtualizes all operating system resources. While commodity operating systems provide virtualization for some resources, complete virtualization requires virtualization for many resources that are already not virtualized,
Table 5.1 – Kernel subsystems and related resources

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process ID</td>
<td>PID and related IDs: thread group, process group, session</td>
</tr>
<tr>
<td>SysV IPC</td>
<td>ID and KEY of message queues, semaphores, shared memory</td>
</tr>
<tr>
<td>Unix IPC</td>
<td>Unix domain sockets, pipes, named pipes</td>
</tr>
<tr>
<td>File system</td>
<td>File system root (chroot)</td>
</tr>
<tr>
<td>Network</td>
<td>Internet domain sockets</td>
</tr>
<tr>
<td>Devices</td>
<td>Device specific resources</td>
</tr>
<tr>
<td>Pseudo-terminals</td>
<td>PTS IDs and devpts pseudo file system</td>
</tr>
<tr>
<td>Pseudo-file systems</td>
<td>E.g. procfs, devpts, shmfs</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Hostname, user/group ID, system name</td>
</tr>
</tbody>
</table>

including process identifiers (PIDs), keys and identifiers for IPC mechanisms such as semaphores, shared memory, message queues, and network addresses. Table 5.1 provides a summary of these additional resources that must be virtualized; a more detailed discussion is presented in [124].

Within this taxonomy of virtualization approaches, complete and host-independent virtualization provides the broadest range of functionality, which includes providing the necessary support for both isolation and migration of applications. An additional distinction between the taxonomies is in the scope of the application with respect to the available systems. Virtualization approaches that are host-dependent and/or partial provide benefits only on a single host, while complete, host-independent virtualization approaches provide the support for applications to migrate and exploit the available systems that are accessible to the entire organization. The remainder of this chapter focuses on the demands of supporting this more general form of virtualization in the context of commodity operating systems.
5.2 Interposition Architecture

To support private virtual namespaces, mechanisms must be provided to translate between the pod’s resource identifiers and the operating system resource identifiers. For every resource accessed by a process in a pod, the virtualization layer associates a virtual name to an appropriate operating system physical name. When an operating system resource is created for a process in a pod, the physical name returned by the system is caught by the virtualization layer, and a corresponding private virtual name is created and returned to the process. Similarly, any time a process passes a virtual name to the operating system, the virtualization layer catches and replaces it with the corresponding physical name. To enable this translation, a mechanism must be employed that redirects the normal control flow of the system so that the private virtual namespaces are employed rather than the default physical namespace.

Interposition is the key mechanism that can provide the requisite redirection needed for virtualization of namespaces. In our context, interposition captures events at the interface between applications and the operating system and performs some processing on those events before passing them down to the operating system or up to the applications. The interposition that needs to be done for implementing operating system virtualization requires some preprocessing to be done before the native kernel functionality is executed, and some post-processing to be done after the native kernel functionality is executed. The interposition implementation itself is accomplished by wrapping the existing system calls with functions belonging to the virtualization layer and translating between virtual names and physical names before and after the original system call is invoked.

System call interposition can be implemented at different layers of the system. We advocate using the loadable kernel module technology that is now available with
all major commodity operating system. A kernel module can provide application-transparent virtualization without base kernel changes and without sacrificing scalability and performance. In addition, by operating in privileged mode, virtualization can provide the security necessary to ensure correct isolation. By working at the level of kernel modules, the virtualization module can utilize the set of exported kernel subroutines, which is a well-defined interface. Using the kernel API also denotes a certain level of portability and stability in the implementation since changes in the kernel API are infrequent, because hundreds of existing libraries also depend on the same API. In other words, virtualization portability is protected to a large extent from kernel changes in a similar way since legacy applications are protected.

There are other approaches to implementing system call interposition. One approach is to implement it as a user-level library [84, 92] such that interposition code is executed in the context of the process running the system call. This is relatively easy to implement, potentially yields more portable code, and utilizes the clear boundary between user-level and kernel-level. Unfortunately, it does not provide effective isolation of applications and can be easily subverted at any time, since the mechanism runs at user-level in the context of the processes. It instead requires their cooperation and does not work for statically-linked libraries or directly executed system calls.

Another approach is to use a kernel process tracing facility such as ptrace [111], which allows a user-level process to monitor another process [174]. The monitoring process is notified of each entry and exit of system calls and receipt of signals, and can read and write the memory of the controlled process. By using available kernel functionality, this process tracing approach can enforce an operating system virtualization abstraction more effectively than strictly user-level approaches. However, ptrace has many limitations in terms of performance and security [174], and the semantics of ptrace are highly system-specific, which results in a non-portable method.
A third approach is to modify the kernel directly to implement interposition. This offers maximum flexibility, with the lowest interposition overhead. However, writing code directly in the kernel is more complicated and cumbersome than in user-level, harder to debug, and the result is most likely to be non-portable. Tying the implementation to the kernel internals requires tracking, in detail, all subsequent kernel updates. Furthermore, imposing a kernel patch, recompilation and reboot process is a serious practical barrier to deployment and ease-of-use.

Given the limitations of other approaches, we have implemented operating system virtualization as a loadable kernel module that works with major Linux kernel versions, including both Linux 2.4 and 2.6 kernels. Our implementation avoids modifications to the operating system kernel, and aims to build strictly on its exported interface as much as possible. It supports the pod abstraction but also allows other processes to run outside of virtualized environments to ease deployment on systems which require such legacy functionality.

### 5.3 Virtualization Challenges

Given this kernel module interposition architecture, we now discuss key implementation challenges in supporting virtualized system calls. Virtualization requires that some state be maintained by the virtualization module. The basic state that needs to be maintained is the pod’s resource names, the underlying system physical resource names, and the mapping between virtual and physical names. Throughout this discussion we emphasize that performance is a primary concern and many of our approaches are engineered to achieve low performance overhead. Table 5.2 provides a summary of the methods and data structures used to maintain virtualization state efficiently.
Chapter 5. Virtual Execution Environment

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System-wide hash table</td>
<td>Convert physical host identifiers to virtual pod identifiers</td>
</tr>
<tr>
<td>Per-pod hash table</td>
<td>Convert virtual pod identifiers to physical host identifiers</td>
</tr>
<tr>
<td>Direct reference</td>
<td>Per-process fast reference to augmented virtualization state</td>
</tr>
<tr>
<td>PID reference count</td>
<td>Protect PIDs of processes inside or outside pods from reuse</td>
</tr>
<tr>
<td>in-pod flag</td>
<td>Indicate that a process is running inside a pod</td>
</tr>
<tr>
<td>init-pending flag</td>
<td>Indicate that a process in a pod is pending initialization</td>
</tr>
<tr>
<td>Outside-pod table</td>
<td>Track identifiers used by processes running outside pods</td>
</tr>
<tr>
<td>Restricted-ID table</td>
<td>Track identifiers without a reference count that are in use</td>
</tr>
<tr>
<td>init-complete flag</td>
<td>Indicate that the virtualization state of a resource is ready</td>
</tr>
<tr>
<td>File system stacking</td>
<td>Virtualize per-pod pseudo file systems view</td>
</tr>
</tbody>
</table>

Table 5.2 – Summary of virtualization methods

A first approximation approach employs two types of hash tables that can be quickly indexed to perform the necessary translation. One is a system-wide hash table indexed by physical identifiers on the host operating system, that returns the corresponding pod and virtual identifier. The other is a per-pod hash table indexed by virtual identifiers specific to a pod, that returns the corresponding physical identifiers. A separate pair of hash tables would be used for each operating system resource that needs to be virtualized, including PIDs, SysV IPC, and pseudo terminals. For multiprocessor and multi-threaded systems, proper hash table maintenance requires locking mechanisms to ensure state consistency. Handling these locks to avoid deadlock and to lower performance overhead is a non-trivial matter, and is discussed in § 5.3.1 on race conditions.

The use of these hash tables alone can result in suboptimal performance. While hash tables provide a constant time lookup operation, there is a non-negligible performance overhead due to added lock contention, extra computation required to do the lookup, and some resulting cache pollution. The system-wide hash table is used for each resource access to determine the pod associated with the running process. The frequent use of this hash table can cause lock contention and impair scalability.
To minimize the cost of translating between pod namespaces and the underlying operating system namespace, we associate with each native process data structure a direct reference to the process’s augmented virtualization state and the process’s pod. These direct references act as a cache optimization that eliminates the need to use the table to access the virtualization data of a process, which in turn reduces the hash table lookup rate.

While this direct association only requires two references, it is unlikely that the native kernel process data structure has two unused references which can be used for this purpose. Instead, an effective solution (that avoids changes to the underlying operating system kernel) is to extend the area occupied on the process’s kernel stack by two pointers that reference the relevant data structures. In this manner, once a kernel process data structure is obtained, there is no need to refer back to any hash tables to translate from physical to virtual identifiers. Because this operation is so common, this reduces the virtualization overhead of the system across a broad range of virtualized system calls and eliminates a major source of lock contention.

For most of the subsystems referenced in Table 5.2, virtualization consists of managing the pod’s state tables and handling the race conditions discussed in § 5.3.1. This includes the following subsystems: PIDs, SysV IPC, network identifiers, and pseudo terminals, though the latter case of pseudo terminals requires further support through the file system. The file system, devices, and most of the miscellaneous systems are accessed through file system operations, whose virtualization specifics are discussed in § 5.3.2. Unix IPC semantics are controlled using file descriptors with process groups, so our handling of fork in § 5.3.1.2 is sufficient handling of Unix IPC virtualization. The remaining subsystem is pseudo file systems which is discussed in § 5.3.3.
5.3.1 Race Conditions

Race conditions occur due to the non-atomic transactions carried out by the system call wrapper subroutines. This is inherent to a virtualization approach that does not modify the kernel and treats kernel subroutines as black-boxes from the virtualization module’s point of view. Race conditions, which make it difficult to maintain consistent virtual state, are more common in multiprocessor and multicore systems, as well as preemptive kernels, but are also present in non-preemptive uniprocessor systems.

Figure 5.1 illustrates the anatomy of a typical system call wrapper. Race conditions can be classified along two axes: where they occur—in the preamble or epilogue of the system call wrapper, and why they occur—caused by identifier initialization, deletion, or reuse.

Preamble races can occur after a resource identifier has been translated from the virtual to the corresponding physical identifier but before the underlying kernel code is invoked. The race occurs if, in this time frame, the resource is released in the kernel, and its physical identifier is subsequently reused and therefore ends up pointing to a distinct resource, possibly in another pod. The race conditions that are due to identifier reuse are generally rare given the very brief vulnerability window in which
an unusual sequence of time consuming events must occur, and because the large size of the namespace from which offending identifiers are drawn. Allocation algorithms typically attempt to avoid reusing a recently reclaimed identifier. Nevertheless, factors such as heavy workload, the presence of swapping activity, having large portions of the namespace already in use, and enabling concurrency can all contribute to the risk of a race event.

*Epilogue races* can occur after the kernel returns a physical identifier of a resource but before the virtualization wrapper converts the physical identifier to a virtual identifier. Epilogue deletion races occur if the resource is freed during the period between kernel invocation and post-processing, which results in its removal from the virtualization state and causes the pending conversion to fail. Epilogue initialization races can appear between the time that a resource is allocated, which is usually after the completion of the underlying kernel code, and the time it is appropriately registered within the virtualization subsystem. Since these two operations are not executed within a single atomic section, the resource instance can be visible and modifiable to processes that do not belong to the same pod, or the resource instance can be exposed prematurely to processes in the same pod. In the following sections, we detail distinct problems and solutions and discuss the applicability of the patterns for other system resources.

### 5.3.1.1 Process ID Races

PID races can occur if a PID is referenced and changes during the execution of a virtualized system call such that stale data ends up being used. A change occurs when the PID is released and reclaimed by the kernel after a process terminates, and the PID may end up being reassigned by the kernel to a newborn process, as seen in Figure 5.2.
Figure 5.2 – PID deletion race: (1) Process A queries the PGID of process B, (2) we convert from virtual to physical, and (3) call the actual kernel system call. If (4) process B now exits, and (5) a new process C gains the same PID, then (6) we convert back from physical to virtual wrongly.

Both `getppid` and `getpgid` are examples of system calls vulnerable to these races. `getppid` does not take arguments so it begins with a trivial preamble, followed by the invocation of the kernel’s system call, and then the epilogue translates the result from the physical value to the virtual one. An epilogue deletion race exists if the mapping of the parent process’s PID changes between the invocation and the translation, which can also occur if the parent process terminates exactly then. One of three effects can occur as a result. First, if the PID is not reused, the kernel system call will return an error. Second, if the PID is reused and assigned to a new process in the same pod, the wrapper will return an erroneous value. Third, if the PID is reused by a new process in another pod, the wrapper will return a meaningless value. Similarly, `getpgid` begins with translating its PID argument from virtual to physical, then
calls the kernel’s system call, and wraps up by translating the return PID back from physical to virtual. Thus, `getpgid` is exposed to the same epilogue deletion race as with `getppid` in addition to a preamble race.

Preamble races are potentially more harmful, especially for system calls that modify process state. For `getpgid`, a preamble race between the preamble and system call invocation can arise if the process terminates exactly then. One of four effects can occur as a result. First, if the PID is not reused, the kernel system call will return an error. Second, if the PID is reused and assigned to a new process in the same pod, the returned value will be the process group ID of another process in the same pod. Both of these cases are harmless as a similar race is inherent to Unix and may legally occur during its normal non-virtualized operation. Third, if the PID is reused and assigned to a process not in a pod, the physical process group ID returned by the kernel will not have a corresponding virtual group ID and the wrapper subroutine will fail. Fourth, if the PID is reused and assigned to a process in some other pod, the process group ID from another pod will be returned. This results in information leakage and violates isolation between pods. It also causes inconsistency as two successive system calls will return different results. Moreover, other system calls that tamper with the system state can result in worse behavior. For instance, a race in the case of `kill` could end up delivering a signal from a process in one pod to another process in some other pod, and `setpgid` could attempt to modify a process group ID of a process in another pod, to a possibly undefined value there.

To prevent these races, we ensure that a reference count on the object in question is taken to guarantee that neither a PID nor the corresponding task structure are freed and reclaimed prematurely. This effectively protects the referenced object in its current state for the duration of the transaction. As long as that reference is held, the kernel will not reclaim the resource even if the process that owns it has exited.
and has subsequently been collected. To implement this, we use the kernel's own reference count primitives for these objects and piggyback on them by calling the corresponding kernel subroutines to modify the reference count.

We minimize the interaction with the kernel by only modifying the kernel's reference count twice during the entire lifetime of a process. It is incremented when the process is associated with a pod, either by entering a pod or as a result of *fork*, and is decremented after the process exits. We combine this with a reference count that is maintained as part of the per process virtualization state. This count is initialized to one when the process joins a pod and modified twice in every transaction that is vulnerable to a PID race. It is incremented at the beginning of the transaction and decremented when the reference is no longer needed. The separation between the kernel's reference count on the original object and the module's reference count on the virtualized object reduces lock contention by preferring per pod locks over the kernel global lock. It also improves portability by reducing the dependency on the kernel without additional code complexity, since the reference count for virtualized objects is also required for other reasons.

Similar to processes in a pod, regular processes not running in a pod are vulnerable to a symmetric race in which a regular process examines another regular process and the latter either enters a pod or dies before the former completes the transaction. If not addressed, this can result in an interaction between a regular process and a process in a pod, which should otherwise be forbidden. For example, consider a regular process that attempts to send a signal to another regular process. If the PID of the latter joins the scope of some pod, either by the owner entering or by being reused after the owner exits, after the sender already completed the PID translation but before it invoked the underlying native *kill*, the sender would end up delivering a signal to a process otherwise invisible to it.
We resolve this race by keeping a reference count for PIDs accessed outside of pods. Regular processes are associated with a special pseudo-pod. Referenced PIDs are then guaranteed not to be reclaimed prematurely. At the same time, processes are prohibited from entering a pod while a positive reference count exists for one of their PIDs. To complete the solution, we added the constraint that a process may only enter a pod while the process executes in its own context. The rationale for this is similar to *fork* and other system calls; a fork cannot be imposed on a process, but rather must be executed by the process itself. This guarantees certain properties on the process state, such it being a well-defined state and a specific entry point, which eliminates a-priori numerous races and other subtleties.

### 5.3.1.2 PID Initialization Races

A key initialization race that must be addressed is correct initialization of the virtualization state of a process, particularly on process creation as a result of the *fork* system call, as seen in Figure 5.3. When *fork* is called, the virtualized system call performs some preliminary internal management and accounting, then invokes the native system call which returns twice, once in the context of the parent, and once in the context of the child. When executed in the context of the child process, the call returns immediately to user-level and does not return control to the kernel, so that no post-processing can be done as part of the virtualized system call execution by the child process. In particular, the child process may return and begin executing before the parent process returns from the native system call.

As a consequence, there is a brief period in time in which the child process can resume execution without informing the virtualization module of its existence. Since the virtualization module is not aware that the child process has already been created, it is not able to initialize any necessary fields in its hash tables for that process,
including any mappings between virtual and physical names for that child. Although
the parent process can attempt to initialize the appropriate data structures on behalf
of the child, it would only do so after its execution of the system call reaches the
post-processing part. There is no guarantee for that to occur before the child process
resumes execution in user-level, and potentially even issues other system calls. The
problem is inherent to a system call interposition approach that treats fork as a
black-box. As a result, it becomes difficult to determine whether the new child process
belongs to a pod and must be virtualized. If the problem is not fixed, the process
will not be isolated within a pod, and may freely interact with the underlying system
and other processes.

In constructing an efficient method to ensure that a child process’s state is properly
initialized, a key observation is that an uninitialized process may execute freely as long

Figure 5.3 – PID initialization race: (1,2) The parent forks and (3) a child is created. The child executes before the parent completes the fork and (4) queries its PID. We (4) call the kernel system call, but (6) cannot convert back from physical to virtual because the virtual PID is uninitialized.
as no interaction occurs with its virtualized state. As soon as such an interaction takes place, the process must first be initialized before it is allowed to continue execution. Assuming a method exists to detect that a process is not initialized, there are three cases to address. First, when a parent process returns from the native `fork` system call, it tests if the child process has already been initialized. If not, it initializes the virtualization state of the child, including storing the mapping of virtual and physical resource names in the appropriate virtualization data structure. Second, the nature of host-independent complete virtualization guarantees that the child process will not access any of its virtualization state until it calls a virtualized system call. As a result, the child process can wait until it calls a virtualized system call to have its virtualization state initialized. Each virtualized system call has a preprocessing step which tests whether the calling process is in a pod and whether its virtualization state has been initialized. When an uninitialized child process executes a virtualized system call, the system notices the uninitialized state and initializes the virtualization state at that time. Third, if some other process attempts to access the uninitialized child process via a virtualized system call, the child process is identified as being uninitialized which causes the system to initialize the virtualization state at that time. Conceptually the solution is to ensure that all direct and indirect access to the resource is virtualized, hence the first time the resource is accessed, it is also initialized.

To provide correct operation with low overhead, we augment the use of hash tables by storing some per process virtualization state as part of the in-kernel process data structure. The data structure used to represent a process in the kernel typically contains a set of flags used to note various process states. In general, the fields in the kernel process structure used to store such information are not completely populated so that unused parts remain. We use two bits of these unused parts to piggyback on
the native `fork` system call to implicitly initialize a minimal virtualization state that identifies a process as being in a pod and uninitialized.

These bits serve as two helper flags for virtualization. The first is the `in-pod` flag and indicates whether a process is in a pod. A crucial advantage of using this field is that it is inherited across child process creation given the semantics of `fork`. A child of a process that is already in a pod atomically inherits the flags and is therefore immediately identified as also being in a pod. Thus, processes in pods can be readily and efficiently filtered and made invisible to regular processes, even when uninitialized, since the flag is inherited. The second flag is the `init-pending` flag and indicates that a process in a pod is pending initialization. A parent process sets its `init-pending` flag when processing the virtualized `fork` system call as part of the preprocessing that occurs before executing the native system call. The flag is inherited atomically by the child process so that both the parent and the child process appear to be uninitialized. The presence of the flag on the parent is meaningless and ignored. The flag is cleared on both when the child process has been initialized.

The combined approach of employing the helper flags in addition to the hash tables provides a performance benefit as well. Hash table lookup is part of the critical path of four common tasks: first, when testing whether a process belongs to a pod when it issues a system call to decide whether to virtualize or not; second, when testing whether a target process (e.g., in a pod) should be masked out from another process (e.g., not in a pod); third, when access to the virtualization data of a process is needed; and fourth, when a physical-to-virtual translation or vice versa is required. Despite their fast lookup times, hash tables may incur non-negligible overhead when used very frequently.

Testing for the flag on a process trivializes the first two tasks and eliminates the need to perform a hash table lookup and thereby improves scalability and performance
by avoiding the need to serialize access to the table, and by eliminating both the extra cycles of the lookup and the associated cache pollution. This is particularly beneficial for regular processes not running in pods that would otherwise suffer a performance degradation due to such a lookup in each virtualized system call. For example, in Algorithm 5.1, a negative results for the \textit{in-pod} test at the beginning of the wrapper short-circuits the \texttt{lookup_pod()} call. Since PID-related functions are a main part of the code path, the overhead for the system as a whole is lowered. Isolation of processes in a pod from processes outside of the pod becomes easy as well: if a process not running in any pod attempts to access a process in a pod, the \textit{in-pod} flag of the process in a pod will already be set and hence it is straightforward to deny access.

5.3.1.3 SysV IPC Races

SysV IPC [159] primitives consist of message queues, shared memory and semaphores. For simplicity, we focus our discussion on message queues, but the same principles apply for the other two primitives. IPC consists of two inter-related resources, namely identifiers and keys. Unlike PIDs, both are global and not associated with specific processes that own a reference to them. Keys identify a context and are persistent
while identifiers are created when such contexts are instantiated to allocate an IPC object, hence representing a specific instance. Once a key has been instantiated, future attempts to instantiate it will resort to the existing instance, until that instance is explicitly deleted. For example, the first call to `msgget` with some key value will allocate a new message queue and assign a unique identifier that represents that key. Subsequent calls will detect the active queue that is associated with the specified key and will return the same identifier, until finally the queue is removed, and so on. IPC identifiers and keys are global resources that must be virtualized using virtual-to-physical and physical-to-virtual hash tables. Like PIDs, they are also mutable during system calls, potentially resulting in initialization, preamble, and epilogue deletion races.

Initialization races can occur when a new identifier is created as it may become visible to the system prior to the initialization of its virtualization data. For instance, an IPC identifier allocated inside a pod whose virtualization state has not yet been initialized cannot be determined to belong to the pod, and therefore may be potentially accessed by a process outside the pod. This issue is aggravated since most IPC primitives will alter the system state, possibly before the resource is ready. Similar to the solution to the process initialization race condition we are assured that the resource cannot be accessed without going through the virtualization layer. Unlike processes, however, the internal data structure that represents IPC resources is not extensible, making it impossible to associate either a pointer or a flag with it.

To prevent misuse of IPC identifiers before their virtualization data is initialized, we introduce a third hash table called the `outside-pod` table to indicate whether a given instance has been initialized. The outside-pod table stores all identifiers in use by processes not in a pod. As with PIDs, this may be thought of as treating the namespace that does not belong to any pod as a pseudo-pod where virtual and physi-
cal identifiers are mapped one-to-one. Regular processes must consult the outside-pod table to access an identifier, analogous to testing the *in-pod* flag for PIDs. They will be blocked from accessing uninitialized identifiers since they will fail to find them in the outside-pod table.

The outside-pod table must be correctly populated to account for IPC resources that may already exist when the virtualization module is loaded. When the module is loaded, it must scan the kernel data structures for instances of IPC objects and place the identifiers in the outside-pod table. Special care must be taken not to overlook an instance that is being created at the time of the scan or afterwards, by a process that started a native (non-virtualized) system call prior to the scan. Otherwise, that identifier will not be accounted for and will consequently become invisible to all processes, including the process that created it. A performance issue with this scheme is the added overhead to IPC related system calls for processes that do not belong to a pod as every operation on an identifier by a process not in a pod implies a lookup in the outside-pod table.

Preamble races can occur due to identifier reuse, as seen in Figure 5.4. For example, consider a process in a pod that holds a valid message queue identifier and calls `msgsnd` to send a message. Suppose that after the translation from the virtual namespace to the physical namespace by the wrapper subroutine, another process in the same pod deletes that message queue from the system, then subsequently that identifier is reused for a new message queue in another pod. When the first process invokes the native system call, it will end up violating the isolation semantics between pods. Since the semantics of IPC allow to remove instances at any time regardless of how many processes may be using them, the kernel does not keep a usage count on them, thus hindering the piggybacking on a native reference count to handle preamble races similarly to PIDs.
<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: \text{SYS_MSGSND}(55, \ldots)</td>
<td>3: \text{SYS_MSGCTL} \newline \hspace{1em}(55, \text{IPCRM},\ldots)</td>
<td>6: \text{SYS_MSGGET}(\ldots) \newline \Rightarrow10</td>
</tr>
<tr>
<td>2: \text{virt_to_phys}(55) \newline \Rightarrow10</td>
<td>4: \text{virt_to_phys}(55) \newline \Rightarrow10</td>
<td></td>
</tr>
<tr>
<td>5: \text{kern_ipcrm}(10) \newline \Rightarrow\text{deleted}</td>
<td></td>
<td>7: \text{kern_msgsnd}(10, \ldots) \newline \Rightarrow\text{ILLEGAL}</td>
</tr>
</tbody>
</table>

**Figure 5.4** – IPC reuse race. (1) Process A sends to queue with virtual ID 55. (2) we convert the ID from physical to virtual. (3,4,5) process B deletes the queue in the same pod, and (6) process C allocates a new queue with the same ID in another pod. When (7) process A calls the actual kernel system call, it sends the message illegally across pod boundary.

To prevent improper reuse of identifiers we propose a fourth hash table named \textit{restricted-ID} table whose purpose is to track all the instances which are being referenced to at any time by the virtualization state together with their reference count. Identifiers will be inserted to the table or referenced in the table during the preamble. They will be taken out if the reference count drops to zero in the epilogue. The virtualization code of \texttt{msgget} will inhibit reuse of an identifier as long as it appears in the table, by having the epilogue inspect new identifiers to ensure that they are not restricted. If they are, the epilogue will deallocate that instance and a new allocation will be attempted.
However, this scheme is still incomplete as illustrated by the following subtlety: suppose a process calls `msgsnd` with some identifier and is preempted between the preamble and the invocation of the actual system call. Suppose also that another process (in the same pod) now removes that instance, and a third process in another pod allocates a new message queue with the same identifier, but is preempted before testing it against the restricted-ID table. If the original process now kicks in it will eventually access the new message queue rather than the intended one. The outcome clearly undermines isolation between pods.

A simple solution is to have the epilogue of `msgget` mark the virtualization state of identifiers that were deallocated so the epilogue of `msgsnd` can detect this condition. When this occurs, `msgsnd` responds by retrying the operation. This is sufficient to ensure that the pod boundaries are respected, since the offending (newly allocated) identifiers never gets a chance to be used in the other pod. Once interaction is confined to the original pod, any side effects are legal since similar circumstances can occur on the native operating system.

A performance issue with the above is the added overhead to IPC related system calls, even for processes that run outside any pod. This overhead comes directly from the extra bookkeeping by the virtualization logic. First, every creation of an instance, either inside or outside a pod, involves a lookup in the restricted-ID table. Second, every operation on an identifier requires that the identifier be inserted in the restricted-ID table for the duration of the system call. Finally, every operation on an identifier by a process not in a pod implies a lookup in the outside-pod table.

Recall that IPC keys identify a context and are persistent, while identifiers represent specific instances of IPC objects. IPC keys are unusual in that the user can select the value of a key when allocating a new message queue. With all other kernel resources, their physical names are assigned solely by the kernel. Since the kernel
always selects unique identifiers, it is not possible for two distinct resources to have the same physical name. In contrast, values of keys are set forth by the application and may potentially coincide across two distinct pods. If the same key is used in two separate pods and passed to the operating system for allocating new message queues, the operating system would not create a message queue in each pod. Instead, it would incorrectly create a single queue and provide the same queue identifier in both pods.

To address this problem, we leverage a special key value, IPC_PRIVATE, designed to allocate private message queues that are not associated with any specific key, and whose identifier cannot be obtained by a subsequent call to msgget. The virtualization wrapper of msgget first searches for the given virtual key in the corresponding hash table and returns the corresponding virtual identifier if found. Otherwise, it invokes the original system call, substituting the original key argument with the special IPC_PRIVATE, causing the operating system to generate a private queue. By creating private message queues, we ensure the uniqueness of each queue within the system. After the system call returns a new identifier, the epilogue allocates a corresponding virtual identifier, populates the table with a new mapping, and associates the virtual key with the virtual ID.

The special behavior of IPC allocation and its virtualization leads to a unique type of preamble race condition. Consider two processes in the same pod trying to allocate a message queue with the same key. Under normal circumstances, the one that is scheduled first will receive the identifier of a new queue, and the other will receive the same identifier. However, if both processes complete their preamble before either of them invokes the real system call, the preamble will have replaced the original key with IPC_PRIVATE and they will now each obtain a distinct, private queue.

The kernel already serializes certain types of IPC calls, like creation, deletion and manipulation of message queues—but not their actual use—with semaphores that ensure
mutual exclusive modifications. Since the offending system calls are already serialized by design, we can use a matching semaphore to protect the virtualization wrapper and make the entire virtualized operation atomic without compromising scalability. This solution also eliminates other race conditions such as epilogue deletion races. A deletion race can only occur when a physical-to-virtual translation of IPC identifiers takes place. In the IPC context this translation only happens during allocation. However, deletion and allocation are mutually exclusive by use of the semaphore and are therefore protected from this race.

5.3.1.4 Pseudo Terminals Races

Pseudo terminals [159] (PTS) are pairs of master/slave devices whose input and output streams are cross linked. The slave end emulates the behavior of a line terminal for the process using it. When the master PTS multiplexer device is opened, a corresponding inode for a slave device is created in the devpts pseudo file system and named /dev/ptsN, where N is the device minor number. The inode is destroyed when the device is released. In addition to virtualizing the pseudo terminal name (the device minor number), it is essential to virtualize the entries in /dev/ptsN to export adequate views in the contexts of different pods. It also prevents races arising from having indirect paths to the resource. This is discussed further in § 5.3.3.

The only conceivable system call involving pseudo terminals is for a process to query the identifier of a terminal attached to a file descriptor. This is always race-free since the initialization of the pseudo terminal must have already been completed when the matching open system call terminated, and the deletion of the pseudo terminal could not have occurred since it would have required that the terminal be closed, yet the file descriptor in question is still kept open. While pseudo terminals are not subject to preamble and deletion races, they are subject to initialization races.
The initialization pitfall is similar to the problem with IPC where a process not in a pod may be able to access a recently created slave device in a pod prior to the completion of its setup. A plausible approach is to use an identical solution to the outside-pod table used with IPC to completely mask out pseudo terminals while not yet initialized.

We employ a lower overhead approach that takes advantage of the ability to borrow one bit from the in-kernel data structure of pseudo terminals. We use the bit to store a per device *init-complete* flag that marks the device as initialized when it is set. Since the bit is unused, the flag is not set upon creation of a pseudo terminal. A newly created pseudo terminal without the flag set is noted as uninitialized and is made invisible to any process trying to access them, be it inside or outside a pod. The flag is set once the initialization is successfully completed and the appropriate associations in the translation hash tables have been made, making the pseudo terminal finally accessible within its pod, or among regular processes if it was originally created outside of any pod. This approach is similar to the *init-pending* flag used for dealing with process initialization races.

Similar to the outside-pod table with SysV IPC, it is possible that access to a pseudo terminal not belonging to any pod is denied for a brief period from processes not in a pod that should otherwise be able to access it. This can happen in the period between the creation of the new inode and the final completion of its initialization. Since such behavior can be expected in traditional Unix, we regard this as a non-virtualization issue. When the virtualization module is loaded, it is necessary to scan the kernel for already open pseudo terminals to set their initialization flags. This must be done carefully to account for rare races that can occur during the scan itself.
5.3.2 File System Virtualization

To provide modular support for multiple file systems, many commodity operating systems provide a virtual file system framework that supports a form of interposition known as file system stacking [186]. We leverage this support along with the `chroot` utility to simplify file system virtualization. File system virtualization is accomplished by creating a special directory per pod that serves as a staging area for the pod’s private file system hierarchy. Storage requirements are minimized by sharing read-only portions of the file system among pods, if applicable, through loopback mounting or networked file systems. The `chroot` system call is used to confine processes that belong to the pod within their private subtree. To ensure that the root of that file system is never traversed, we use a simple form of file system stacking that overloads the underlying file system permission function to implement a barrier directory that enforces the `chroot`-ed environment and ensures that it is only accessible to files within the owner pod. This use of file system stacking leverages existing kernel functionality and avoids the need to replicate that functionality as part of the virtualization implementation.

5.3.3 Pseudo File Systems

Pseudo file systems are memory-based file systems that provide the user with an interface to kernel resources and facilities. Pseudo file systems share three key properties. First, they provide a public, indirect path to a view of global resources. Second, creation and deletion of resource instances are reflected dynamically in this view because the actual behavior of the resources is tracked using dedicated callback subroutines. Third, they may generate a process specific view that is context dependent and differs among processes (e.g., the symbolic link `/proc/self`). To ensure proper virtualiza-
tion semantics, we must virtualize these views to provide context dependent views corresponding to the respective pod being used. We briefly discuss how this is done cleanly and simply for two important pseudo file systems, devpts and procfs.

The devpts file system provides an interface to pseudo terminals. Similar to the file system virtualization described earlier, we use a file system stacking approach to virtualize devpts. Given that each pod uses a dedicated subtree of the file system as its root file system, we provide a pod devpts by stacking an instance of a virtualization file system on the /dev/pts directory in each pod. This is a very lightweight stackable file system whose sole purpose is to virtualize the underlying file system in the context of the specific pod in which it resides. In addition, the required logic is completely independent of the specific pseudo file system, which significantly reduces complexity while leveraging the generality and portability of file system stacking.

The procfs file system maintains a view of the processes running in the system and of an array of system properties. A key feature is that it provides an exported interface that loadable kernel modules can use to dynamically extend its layout. We harness this dynamic extensibility to provide each pod with the requisite context dependent view. For each pod, the virtualization layer automatically creates a private subtree within the procfs hierarchy by mirroring the original file system structure. To keep the overhead low, we do not replicate code or create additional inodes, but instead use hardlinks to refer to existing inodes. This subtree is loopback-mounted at the appropriate point (/proc) within the pod’s root subtree. This approach is appealing due to the simplicity and lowered virtualization overhead compared to other approaches such as file system stacking.
5.4 Evaluation

To demonstrate the effectiveness of our operating system virtualization implementation, we measured its performance on both micro-benchmarks and real application workloads running in both uniprocessor (UP) and multiprocessor (SMP) systems. We quantify the overhead of operating system virtualization, measure the additional cost of correctly handling various race conditions, and also present measurements using hardware virtualization to provide a basis of comparison between virtualization approaches.

We performed our experiments on four configurations: Base—a vanilla Linux system providing a baseline measure of system performance; With–Base with our virtualization module loaded but no pods instantiated, providing a measure of the overhead incurred by regular processes running outside of a pod; Pod–Base with our virtualization module loaded and benchmarks executed inside of a pod, providing a measure of the overhead incurred when running inside of a pod; Xen—a hardware virtualization system with benchmarks executed inside of a virtual machine running a Linux operating system, providing a measure of the overhead of hardware virtualization. Xen was used given its claims of superior performance versus other hardware virtualization systems [47]. The Xen VM was configured with 512 MB RAM unless otherwise indicated. We tested the small RAM allocation for Xen and found that it did not adversely affect the performance of Xen benchmarks. This fact was assessed by providing the Xen domain with increasingly more RAM, which resulted in no appreciable performance gains.

We ran our experiments on both Linux 2.4 and 2.6 kernels, which represent the two major versions of Linux still in use today, but only present data for Linux 2.6.11 for brevity. The operating system configuration used was Debian Stable and we used the
latest version of Xen available with Debian at the time of our experiments, version 2.0.6. For Base, With, and Pod, we present results for Linux with SMP disabled and enabled; Xen 2.0 does not support multiprocessor VMs. We conducted the measurements on an IBM HS20 eServer BladeCenter. Each blade had dual 3.06 GHz Intel Xeon\textsuperscript{TM} CPUs, 2.5 GB RAM, a 40 GB local disk, and was connected to a Gigabit Ethernet switch. Hyperthreading was enabled for the SMP kernel measurements. To provide a fair comparison, we ensured that memory size was not a limitation for any of our experiments.

5.4.1 Micro-benchmarks

To measure the basic cost of our operating system virtualization prototype, we used micro-benchmarks to measure the performance of different system calls with different types of virtualization overhead. Seven micro-benchmarks were used. Each ran a system call in a loop and measured its average execution time: \texttt{getpid}, \texttt{getsid}, \texttt{getpgid}, \texttt{fork}, \texttt{execve}, \texttt{shmget}, and \texttt{shmat}. In particular, the \texttt{fork} benchmark forked and waited for a child that exited immediately, the \texttt{execve} benchmark forked and waited for a child that executed a program that exits immediately, the \texttt{shmget} benchmark created and removed IPC shared memory segments, and the \texttt{shmat} benchmark attached a segment of shared memory twice, modified both copies, and detached it.

Figure 5.5 shows the execution time of each benchmark on each configuration using a UP kernel and normalized to \textit{Base}. Each bar shows the actual execution time reported by the benchmark. Comparing \textit{With} and \textit{Pod} with \textit{Base}, the measurements show that operating system virtualization overhead is quite small in all cases, generally 1-2% and less than 15% in the worst case. Comparing \textit{Xen} with \textit{Base}, the
measurements show that hardware virtualization overhead is much larger especially for more complex system calls, almost five times larger for \texttt{fork} and \texttt{shmat} and more than two times larger for \texttt{execve} and \texttt{shmget}. The only system calls on which \textsc{Xen} did not incur substantial overhead were simple PID-related system calls. These measurements show that operating system virtualization can provide much less overhead than hardware virtualization.

We performed a simple comparison on the UP kernel to quantify the cost of using \texttt{ptrace} for system call interposition in lieu of our kernel module implementation. We executed a version of the \texttt{getpid} micro-benchmark that consists of two processes: a tracer and a tracee. The tracer process is notified about every entry and exit of system calls of the tracee. It then peeks into the tracee’s memory, emulating the work of the wrapper’s preamble and epilogue. The tracee executes \texttt{getpid} repeatedly and we measured the average execution time for the system call. The average execution time was $5.5 \mu s$ for tracing without peeking and $7.7 \mu s$ for tracing and peeking. \texttt{getpid}
degrades by a factor of 13 just for monitoring, and by a factor of 20 if the tracer also peeks into the tracee’s memory. This overhead does not even take into account the added cost of basic virtualization functionality. In comparison, our kernel module virtualization overhead is only 2% for the same system call.

We also compared our measurements with results reported for in-kernel interposition mechanisms. SLIC [65] reports 10% overhead due to the basic dispatcher code, roughly 35% for the interception of \texttt{getpid}, and somewhat lower for more involved system calls. However their base system was a slower UltraSparc and we expect their overhead would be much less on a more modern system. Systrace [139] reports 30% overhead for the same case, which includes their security policy checks, on hardware that is similar to ours. Our measurements suggest that a loadable kernel module implementation is not outperformed by an implementation that modifies the kernel directly.

The actual \texttt{With} and \texttt{Pod} execution times for the first three benchmarks shown in Figure 5.5 provide a quantitative measure of the basic cost of operating system virtualization functionality. \texttt{getpid} only adds a test of the in-pod and init-pending flags to determine whether the process is in a pod, and if so, whether it is pending initialization. The overhead is 8 ns and represents the minimum overhead of a virtualized system call. \texttt{With} and \texttt{Pod} execution times are the same for \texttt{getpid} because the cost of obtaining the virtual identifier is negligible since it is stored as part of the per process virtualization state, which is directly referenced; no hash table translation is required. In addition to testing the in-pod and init-pending flags, \texttt{getsid} requires using the hash table to translate the return value from a physical to virtual identifier if a process is in a pod. This translation overhead is 14 ns as indicated by the difference between \texttt{With} and \texttt{Pod} \texttt{getsid} execution times. In addition to testing the in-pod and init-pending flags, \texttt{getpgid} needs to do a hash table lookup and modify
the kernel’s process reference count for the With case, as discussed in § 5.3.1.1. This added 13 ns versus the Base case due to the lookup, since the reference count adds negligible overhead. For the Pod case, getpgid also needs to modify the virtualization module’s process reference count and perform an additional hash table translation, adding 15 ns versus the With case.

While more complex system calls require more operating system virtualization logic, the overhead of the additional logic is amortized by the additional overhead of the native system call. For example, even though the virtualization cost of fork is almost a microsecond as shown in Figure 5.5, the virtualization overhead as a percentage of the system call execution time is only 1%. The overhead is due to allocating and preparing the virtualization data structure for the child process, linked list maintenance and ensuring correct initialization. All of the more complex system calls have small overhead except for shmget, which has 15% overhead for With and Pod compared to Base. The higher overhead here is largely due to the use of an additional semaphore as discussed in § 5.3.1.3, which does not compromise scalability, but it does increase execution time.

Figure 5.6 shows the execution time of each benchmark on each configuration using a SMP kernel and normalized to Base. Each bar shows the actual execution time reported by the benchmark. No Xen results are shown due to its lack of SMP support in the version that we used. Comparing With and Pod with Base, the measurements show that operating system virtualization overhead is quite small in all cases, generally 2-4% and less than 35% in the worst case.

Compared to Figure 5.5, the measurements show that most of the micro-benchmarks take longer to run when using the Base SMP kernel versus the Base UP kernel. This is due to the kernel’s locking mechanisms, which are trivialized in the UP kernel. Operating system virtualization overhead is small for most system calls in the SMP
case, but is noticeably higher for three of the micro-benchmarks, `getsid`, `getpgid`, and `shmget`. In those cases, the extra locking mechanisms required by the virtualization module result in more overhead for processes running in a pod. The cost of these mechanisms is more prominent for the simple `getsid` and `getpgid` calls but is amortized for the more complex system calls. However, the cost of synchronization is noticeable for `shmget` because it requires the use of a semaphore which is much more expensive than simple spin locks. Except for `getsid`, `getpgid`, and `shmget`, operating system virtualization overhead was roughly 3% or less in all other cases.

We also measured virtualization overhead using LMbench [112] to provide an additional comparison of operating system and hardware virtualization costs. Figure 5.7 shows LMbench local communication and VM system latency measurements for the UP kernel normalized to `Base`; SMP results are not presented given Xen’s lack of SMP support. Each bar also shows the actual benchmark execution time. `Pod` measurements show negligible overhead in almost all cases compared to `Base`. In contrast,
Xen measurements show significant overhead in all cases, ranging between 57%–370% extra overhead compared to Base. These measurements show operating system performance but do not necessarily involve system calls. As a result, operating system virtualization overhead is negligible compared to hardware virtualization overhead.

5.4.2 Application Benchmarks

To provide a more realistic measure of virtualization cost that is expected in actual use, we now turn our attention to real application workloads. While virtualization is pivotal to DejaView’s functionality, the ability to checkpoint and revive applications and to encapsulate applications in isolated environments, in a manner that is transparent and lightweight, is beneficial in a broader context. We evaluate both desktop and server applications to show the overhead for both the DejaView use case as well as other potential use cases that involve servers, such as supporting server consolida-
tions and improving system security. Because server applications are generally more demanding than interactive desktop applications that often allow idle periods during user think times, results for server applications provide a conservative performance measure in the context of DejaView.

We measured the performance of different virtualization approaches using five different application workloads: make—complete build of the Linux kernel using gcc 3.3 (make -j 10); hackbench—a scheduler performance scalability benchmark which creates many processes in groups of readers and writers sending small messages [76] (32 groups); mysql—Super Smack 1.3 database benchmark using MySQL 4.1.9; volano—VolanoMark Java chat server benchmark 2.5.0.9 using Blackdown Java 2 Runtime Environment 1.4.2-02; httperf—httperf 0.8 web server performance benchmark using Apache web server 2.0.53.

Figure 5.8 shows the execution time of the make, hackbench, mysql, and volano benchmarks on Base and Pod using both UP and SMP kernels, and on Xen using a UP kernel, with all measurements normalized to UP Base. (We omit the With case, since the micro-benchmarks results already show that the overhead for regular processes running outside a pod is negligible.) Each bar shows the actual execution time reported by the benchmark. The mysql benchmark reports queries per second, but is converted to query service time to be consistent with the other benchmarks.

The measurements show that Xen has the worst performance on all of the applications, incurring more than 40% overhead in the worst case. In contrast, the measurements show that Pod provides comparable performance to Base across both UP and SMP kernels. Operating system virtualization overhead as shown by the Pod case is 2% or less for all applications. Furthermore, the SMP measurements show that operating system virtualization directly benefits from SMP support in the underlying kernel and can take advantage of available SMP hardware to improve the
performance of multi-process and multi-threaded applications. The SMP results for operating system virtualization show a 1.5 to 2.2 speedup versus using the UP kernel.

To provide a measure of performance scalability, we measured the performance of httperf as we scaled the number of instances of the benchmark running at the same time. For Base, we ran multiple instances of Apache with each instance listening on a different port. For Pod, we ran an Apache instance in each pod. We scaled the number of httperf and Apache instances from 1 to 128 and measured the average request rate across all instances.

We also executed this benchmark inside a Xen virtual machine running a Linux operating system, to provide a measure of the overhead of hardware virtualization. Xen was used given its claims of superior performance versus other hardware virtualization systems [47]. For Xen, we ran an Apache instance in each Xen VM. To enable Xen to scale to a larger number of Apache instances, we configured the Xen VM used with 128 MB RAM.
Figure 5.9 shows the results of this experiment. As expected, the average httpperf performance per instance decreases as the number of instances increases due to competition for a fixed set of hardware resources. httpperf performed similarly on all systems when only one instance was executed. However, as the number of instances increased, httpperf performance on Xen falls off substantially compared to its performance on Base or Pod. Xen scalability was further limited by its inability to run more than 16 application instances at the same time because it could not allocate any additional VM instances given the 128 MB RAM per VM and the 2 GB RAM available in the machine used. In contrast, both Base and Pod continue to provide comparable scalable performance up to 128 instances. It is worth noting that in addition to performance scalability, Xen is also limited by storage scalability since each VM requires a separate operating system image. Since operating system virtualization does not require separate operating system images per virtual execution environment, it does not suffer from this storage limitation.
5.5 Summary

In this chapter, we presented DejaView’s virtual execution environment which relies on operating system virtualization to provide private virtual namespaces for computing sessions, a prerequisite for live execution recording described in the next chapter. The construction of such a framework presents a myriad of traps and pitfalls, even for the most cautious developer, that if overlooked may result in a weak, incomplete virtualization. We explored in depth key implementation challenges in supporting operating system virtualization in the context of commodity operating systems, including system call interposition, virtualization state management, and race conditions. Using an operating system virtualization prototype that works across two major versions of Linux (namely, Linux 2.4 and 2.6 series kernels) we demonstrated the benefits of a loadable kernel module implementation and showed that the overhead of this approach is substantially less than other approaches. Our experimental results on both 2.4 and 2.6 kernels on both uniprocessor and multiprocessor systems demonstrate that our approach can provide fine-grain virtualization with very low overhead.
Chapter 6

Live Execution Recording

In this chapter, we present DejaView’s live execution recording approach and the mechanisms available to revive a past session from recorded data. DejaView leverages the virtual execution environment presented in Chapter 5 to transparently encapsulate the entire user’s desktop session, including its virtual display server, in a private virtual namespace. Decoupling applications and display state from the underlying operating system allows them to be continuously checkpointed and later revived in a consistent manner. Operating above the operating system instance to encapsulate the desktop session only, not an entire machine instance, provides fast operation by only saving applications state, not the entire operating system instance.

DejaView’s recording allows users to revive their desktop as it was at any point in the past, for cases where “static” visual information is not enough. Revived sessions behave just like the main desktop session; the user is free to continue to interact with them and possibly diverge from the path taken in the original recording. Furthermore, the sessions peacefully coexist with each other, as the virtual execution environment surrounding each session guarantees that underlying resources will be safely multiplexed and the sessions will be completely isolated from each other.
DejaView combines a coordinated checkpoint with 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 that corresponds to the time at which the checkpoint was taken for simultaneous read-write usage. DejaView’s recording benefits from several optimizations aimed to reduce application downtime during checkpoint, and from a checkpoint policy for throttling of the checkpointing rate that reduces both the runtime impact of checkpoints on interactive applications and the storage requirements.

6.1 Application Checkpoint-Restart

Application checkpoint-restart is the ability to save the state of a running application to secondary storage so that it can later resume its execution from the time at which it was checkpointed. Checkpoint-restart can provide many potential benefits, including fault recovery by rolling back an application to a previous checkpoint, better application response time by restarting applications from checkpoints instead of from scratch, and improved system utilization by stopping long running computationally intensive jobs during execution and restarting them when load decreases. An application can be migrated by checkpointing it on one machine and restarting it on another, which provides further benefits, including fault resilience by migrating applications off of faulty hosts, dynamic load balancing by migrating applications to less loaded hosts, and improved service availability and administration by migrating applications before host maintenance so that applications can continue to run with minimal downtime.

Many important applications consist of multiple cooperating processes. In order to checkpoint-restart these applications, not only must application state associated with
Chapter 6. Live Execution Recording

Each process be saved and restored, but the state saved and restored must be globally consistent and preserve process dependencies. Operating system process state including shared resources and various identifiers that define process relationships such as group and session identifiers must be saved and restored correctly. Furthermore, for checkpoint-restart to be useful in practice, it is crucial that it transparently support the large existing installed base of applications on commodity operating systems.

Zap [124] is a system that provides transparent checkpoint-restart of unmodified applications that may be composed of multiple processes on commodity operating systems. The key idea is to introduce a thin virtualization layer on top of the operating system that encapsulates a group of processes in a virtualized execution environment and decouples them from the operating system. This layer presents a host-independent virtualized view of the system so that applications can make full use of operating system services and still be checkpointed and then restarted at a later time on the same machine or a different one. While previous work presents key aspects of Zap’s design, it did not describe a number of important engineering issues in building a robust checkpoint-restart system. In particular, a key issue that was not discussed is how to transparently checkpoint multiple processes such that they can be restarted in a globally consistent state.

Building on Zap, we address this consistency issue and discuss in detail key design issues in building a transparent checkpoint-restart mechanism for multiple processes on commodity operating systems. We combine a kernel-level checkpoint mechanism with a hybrid user-level and kernel-level restart mechanism to leverage existing operating system interfaces and functionality as much as possible for checkpoint-restart. We introduce a novel algorithm for accounting for process relationships that correctly saves and restores all process state in a globally consistent manner. This algorithm is crucial for enabling transparent checkpoint-restart of interactive graphical applica-
tions and correct job control. We introduce an efficient algorithm for identifying and accounting for shared resources and correctly saving and restoring such shared state across cooperating processes. To reduce application downtime during checkpoints, we also provide a copy-on-write mechanism that captures a consistent checkpoint state and correctly handles shared resources across multiple processes.

6.1.1 Virtualization Support

To enable checkpoint-restart, we leverage our virtual execution environment to decouple applications from the underlying host. Checkpoint-restart operates on the pod abstraction, which is used to encapsulate a set of processes in a self-contained unit that can be isolated from the system, checkpointed to secondary storage, and transparently restarted later. This is made possible because each pod has its own virtual private namespace, which provides the only means for processes to access the underlying operating system.

Operating system resource identifiers, such as process IDs, must remain constant throughout the life of a process to ensure its correct operation. However, when a process is checkpointed and later restarted, possibly on a different operating system instance, there is no guarantee that the system will provide the same identifiers to the restarted process; those identifiers may in fact be in use by other processes in the system. The pod namespace addresses these issues by providing consistent, virtual resource names. Names within a pod are trivially assigned in a unique manner in the same way that traditional operating systems assign names, but such names are localized to the pod. These virtual private names are not changed when a pod is restarted to ensure correct operation. Instead, pod virtual resources are transparently remapped to real operating system resources.
Since the namespace is virtual, there is no need for it to change when the pod is migrated, ensuring that identifiers remain constant throughout the life of the process, as required by applications that use such identifiers. Since the namespace is private to a given pod, processes within the pod can be checkpointed and restarted as a group, while avoiding resource naming conflicts among processes in different pods.

In addition to providing a private virtual namespace for processes in a pod, our virtualization approach provides three key properties so that it can be used as a platform for checkpoint-restart. First, it provides mechanisms to synchronize the virtualization of state across multiple processes consistently with the occurrence of a checkpoint, and upon restart. Second, it allows the system to select predetermined virtual identifiers upon the allocation of resources when restarting a set of processes so that those processes can reclaim the same set of virtual resources they had used prior to the checkpoint. Third, it provides virtualization interfaces that can be used by checkpoint and restart mechanisms to translate between virtual identifiers and real operating system resource identifiers. During normal process execution, translating between virtual and real identifiers is private to the virtualization layer. However, during checkpoint-restart, the checkpoint and restart functions may also need to request such translations to match virtual and real namespaces.

We combine pod virtualization with a mechanism for checkpointing and restarting multiple cooperating processes in a pod. For simplicity, we describe the checkpoint-restart mechanism assuming a shared storage infrastructure across participating machines. In this case, file system state is not generally saved and restored as part of the pod checkpoint image to reduce checkpoint image size. Available file system snapshot functionality [24, 53, 74, 117] can be used to also provide a checkpointed file system image. We focus only on checkpointing process state; details on how to checkpoint file system, network, and device state are beyond the scope of this dissertation.
6.1.2 Key Design Choices

Before describing the checkpoint-restart mechanisms in further detail, we first discuss three key aspects of their overall structure: first, whether the mechanism is implemented at kernel-level or user-level; second, whether it is performed within the context of each process or by an auxiliary process; third, the ordering of operations to allow streaming of the checkpoint data.

**Kernel-level vs. user-level:** Checkpoint-restart is performed primarily at kernel-level, not at user-level. This provides application transparency and allows applications to make use of the full range of operating system services. The kernel-level functionality is explicitly designed so that it can be implemented as a loadable module without modifying, recompiling, or relinking the operating system kernel. However, to simplify process creation, a necessary step for restarting from a checkpoint, we leverage existing operating system services to perform the process creation phase of the restart algorithm at user-level. The standard process creation system call `fork` is used to reconstruct the process forest. Combining kernel-level checkpoint with a hybrid user-level and kernel-level restart mechanism leverages existing operating system interfaces and functionality as much as possible.

**In context vs. auxiliary:** Processes are checkpointed from outside of their context and from outside of the pod using a separate auxiliary process, but processes are restarted from inside the pod within the respective context of each process.

Checkpoint is executed by an auxiliary process that runs outside of the pod for two reasons. First, since all processes within the pod are checkpointed, this simplifies the implementation by avoiding the need to special case the auxiliary process from being checkpointed. Second, the auxiliary process needs to make use of unvirtualized
operating system functions to perform parts of its operation. Since processes in a pod are isolated from processes outside of the pod when using the standard system call interface [124], the auxiliary process uses a special interface for accessing the processes inside of the pod to perform the checkpoint.

Furthermore, checkpoint is done not within the context of each process for four reasons. First, using an auxiliary process makes it easier to provide a globally consistent checkpoint across multiple processes by simply quiescing all processes then taking the checkpoint; there is no need to have each process run to checkpoint itself, nor attempt to synchronize their checkpoint execution. Second, a set of processes is allowed to be checkpointed at any time and not all of the processes may be runnable. If a process cannot run, for example if it is stopped at a breakpoint as a result of being traced by another process, it cannot perform its own checkpoint. Third, to have checkpoint code run in the process context, the process must invoke this code involuntarily since we do not require process collaboration to checkpoint. While this can be addressed by introducing a new specific signal to the kernel [102] and arranging for this signal to be served within the kernel, it requires kernel modifications and cannot be implemented by a kernel module. Fourth, it allows for using multiple auxiliary processes concurrently (with simple synchronization) to accelerate the checkpoint operation.

Unlike checkpoint, restart is done within the context of the process that is restarted for two reasons. First, operating within process context allows us to leverage the vast majority of available kernel functionality that can only be invoked from within that context, including almost all system calls. Although checkpoint only requires saving process state, restart is more complicated as it must create the necessary resources and reinstate their desired state. Being able to run in process context and leverage available kernel functionality to perform these operations during restart significantly
simplifies the restart mechanism. Second, because the restart mechanism creates a new set of processes that it completely controls on restart, it is simple to cause those processes to run, invoke the restart code, and synchronize their operations as necessary. As a result, the complications with running in process context during checkpoint do not arise during restart.

More specifically, restart is done by starting a supervisor process which creates and configures the pod, then injects itself into the pod. Once it is in the pod, the supervisor forks the processes that constitute the roots of the process forest. The root processes then create their children, which recursively create their descendants. Once the process forest has been constructed, all processes switch to operating at kernel-level to complete the restart. The supervisor process first restores globally shared resources, then each process executes concurrently to restore its own process context from the checkpoint. When all processes have been restored, the restart completes and the processes are allowed to resume normal execution.

**Data streaming:** The steps in the checkpoint-restart algorithm are ordered and designed for streaming to support their execution using a sequential access device. Process state is saved during checkpoint in the order in which it needs to be used during restart. For example, the checkpoint can be directly streamed from one machine to another across the network and then restarted. Using a streaming model provides the ability to pass checkpoint data through filters, resulting in extremely flexible and extensible architecture. Example filters include encryption, signature/validation, compression, and conversion between operating system versions.
6.2 Record

DejaView records the desktop session by frequently checkpointing all of the processes associated with the session, which consist of all processes encapsulated in the virtual execution environment. 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 [29] 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.

6.2.1 Consistent Checkpoints

DejaView runs a privileged process outside of the user’s virtual execution environment to perform a consistent checkpoint of the session as follows. 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. This design allows to checkpoint at any time, even when not all processes are runnable.

More specifically, a checkpoint is performed in the following steps:

1. **Quiesce pod**: To ensure that a globally consistent checkpoint is taken of all the processes in the pod, the processes are quiesced. This forces the processes to transfer from their current state to a controlled standby state to freeze them for the duration of the checkpoint.

2. **Record pod properties**: Record pod configuration information, in particular file system configuration information indicating where directories private to the pod are stored on the underlying shared storage infrastructure.

3. **Dump process forest**: Record process inheritance relationships, including parent-child, sibling, and process session relationships.

4. **Record globally shared resources**: Record state associated with shared resources not tied to any specific process, such as System V IPC state, pod’s network address, hostname, system time and so forth.

5. **Record process associated state**: Record state associated with individual processes and shared state attached to processes, including process run state, program name, scheduling parameters, credentials, blocking and pending signals, CPU registers, FPU state, `ptrace` state, file system namespace, open files, signal handling information, and virtual memory.
6. **Snapshot file system**: Record a snapshot of the file system state to ensure a consistent file system view corresponding to the checkpoint.

7. **Continue pod**: Resume the processes in the pod once the checkpoint state has been recorded to allow the processes to continue executing, or terminate the processes and the pod if, for instance, the checkpoint is being used to migrate the pod to another machine.

8. **Commit data**: Write out buffered recorded data (if any) to storage (or to the network) and optionally force flush of the data to disk.

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` [166], LVM [105], 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 [91], 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.

6.2.2 Optimize 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. Furthermore, to avoid the cost of swapping, the memory pages of the designated buffer are pinned to main memory.

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 [56, 129] 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 [129, 140, 164] 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 [61].

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, such as `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.

By using the standard API for memory protection, DejaView’s incremental checkpoint mechanism is relatively straightforward and simple to implement. An alternative approach would be to interpose on the operating system’s page-fault handler, but providing a similar mechanism at the page level is extremely involved and requires intimate knowledge of the operating system internals, which can be subject to frequent changes as newer releases are available. It would also incur significant performance overhead since it places additional conditional code in a common critical path of the entire memory subsystem.

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.

6.2.3 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
Chapter 6. Live Execution Recording

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 (for example, during 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 [125], 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. Alternatively, a user may reduce the checkpoint frequency when display activity decreases but user input is present.

6.3 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 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, \texttt{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.

More specifically, complementary to the checkpoint, a restart is performed in the following steps:

1. \textit{Restore pod properties}: Create a new pod, read pod properties from the checkpoint image and configure the pod, including restoring its file system configuration.

2. \textit{Restore file system state}: Provide a read-write file system view that corresponds to the snapshot taken at the time of the checkpoint.

3. \textit{Restore process forest}: Read process forest information, create processes at the roots of the forest, then have root processes recursively create their children.

4. \textit{Restore globally shared resources}: Create globally shared resources, not tied to any specific process, including creating the necessary virtual identifiers for those resources.

5. \textit{Restore process associated state}: Each created process in the forest restores its own state then quiesces itself until all other processes have been restored.

6. \textit{Continue}: Once all processes in the pod are restored, resume them so they can continue execution.
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. Each incremental checkpoint contains all the state related to the process forest and shared resource, but only a partial view of the memory state of processes. 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.

6.3.1 File System Restore

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 [183] 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 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.

For example, if both layers contain a file \texttt{bar}, only the top most layer’s version of the file is visible. To provide a consistent semantic, if a file is deleted, a “white out” mark is also created on the top most layer to ensure that files existing on a lower layer are not revealed. To continue the example, if the file \texttt{bar} were deleted, it would not allow the \texttt{bar} on the lower layer to be revealed.

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.

### 6.3.2 Network Connectivity

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 \texttt{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.

6.4 Quiescing Processes

Quiescing a pod is the first step of the checkpoint, and is also the last step of the
restart as a means to synchronize all the restarting processes and ensure they are all
completely restored before they resume execution. Quiescing processes at checkpoint
time prevents them from modifying system state, and thus prevents inconsistencies
from occurring during the checkpoint. Quiescing also puts processes in a known state from which they can easily be restarted. Without quiescing, checkpoints would have to capture potentially arbitrary restart points deep in the kernel, wherever a process might block.

Processes are quiesced by sending them a SIGSTOP signal to force them into the stopped state. A process is normally either running in userspace or executing a system call in the kernel, in which case it may be blocked. Unless we allow intrusive changes to the kernel code, signaling a process is the only method to force a process from userspace into the kernel or to interrupt a blocking system call. The SIGSTOP signal is guaranteed to be delivered and not ignored or blocked by the process. Using signals simplifies quiescing as signal delivery already handles potential race conditions, particularly in the presence of threads.

Using SIGSTOP to force processes into the stopped state has additional benefits for processes that are running or blocked in the kernel, which will handle the SIGSTOP immediately prior to returning to user mode. If a process is in a non-interruptible system call or handling an interrupt or trap, it will be quiesced after the kernel processing of the respective event. The processing time for these events is generally small. If a process is in an interruptible system call, it will immediately return and handle the signal. The effect of the signal is transparent as the system call will in most cases be automatically restarted, or in some cases return a suitable error code that the caller should be prepared to handle. The scheme is elegant in that it builds nicely on the existing semantics of Unix/Linux, and ideal in that it forces processes to a state with only a trivial kernel stack to save and restore on checkpoint-restart.

In quiescing a pod, we must be careful to also handle potential side effects [159] that can occur when a signal is sent to a process. For example, the parent of a process is always notified by a signal when either SIGSTOP or SIGCONT signals are handled.
by the process, and a process that is traced always notifies the tracer process about every signal received. While these signals can normally occur on a Unix system, they may have undesirable side effects in the context of checkpoint-restart. For example, a debugger will make such signals visible to the user if the debugging session is quiesced. We address this issue by ensuring that the virtualization layer masks out signals that are generated as a side effect of the quiesce and restore operations.

The use of **SIGSTOP** to quiesce processes is sufficiently generic to handle every execution scenario with the exception of three cases in which a process may already be in a state similar to the stopped state. First, a process that is already stopped does not need to be quiesced, but instead needs to be marked so that the restart correctly leaves it in the stopped state instead of sending it a **SIGCONT** to resume execution.

Second, a process executing the **sigsuspend** system call is put in a deliberate suspended state until it receives a signal from a given set of signals. If a process is blocked in that system call and then checkpointed, it must be accounted for on restart by having the restarting process call **sigsuspend** as the last step of the restart, instead of stopping itself. Otherwise, it will resume to user mode without really having received a valid signal.

Third, a process that is traced via the **ptrace** mechanism [111] will be stopped for tracing at any location where a trace event may be generated, such as entry and exit of system calls, receipt of signals, events like fork, vfork, exec, and so forth. Each such trace point generates a notification to the controlling process. The ptrace mechanism raises two issues. First, a **SIGSTOP** that is sent to quiesce a pod will itself produce a trace event for traced processes, which –while possible in Unix– is undesirable from a look-and-feel point of view (imagine your debugger reporting spurious signals received by the program). This is solved by making traced process exempt from quiesce (as they already are stopped) and from continue (as they should remain stopped). Second,
like sigsuspend, the system must record at which point the process was traced, and use this data upon restart. The action to be taken at restart varies with the specific trace event. For instance, for system call entry, the restart code will not stop the process but instead cause it to enter a ptrace-like state in which it will block until told to continue. Only then will it invoke the system call directly, thus avoiding an improper trigger of the system call entry event.

### 6.5 Process Forest

To checkpoint multiple cooperating processes, it is crucial to capture a globally consistent state across all processes, and preserve process dependencies. Process dependencies consist of process hierarchy such as parent-child relationships, identifiers that define process relationships such as Unix process group and session identifiers (PGIDs and SIDs respectively) [159], and shared resources such as common file descriptors. The first two are particularly important for interactive applications and other activities that involve job control. All of these dependencies must be checkpointed and restarted correctly. The term process forest encapsulates these three components: hierarchy, relationships and resources sharing. On restart, the restored process forest must satisfy all of the constraints imposed by process dependencies. Otherwise, applications may not work correctly. For instance, incorrect settings of SIDs will cause incorrect handling of signals related to terminals (including xterm), as well as confuse job control since PGIDs will not be restored correctly either.

A useful property of our checkpoint-restart algorithm is that the restart phase can recreate the process forest using standard system calls, simplifying the restart process. However, system calls do not allow process relationships and identifiers to be changed arbitrarily after a process has already been created. A key observation in
devising suitable algorithms for saving and restoring the process forest is determining what subset of dependencies require a priori resolution, then leaving others to be setup retroactively.

There are two primary process relationships that must be established as part of process creation to correctly construct a process forest. The key challenge is preserving Unix session relationships\(^1\). Sessions must be inherited by correctly ordering process creation because the operating system interface only allows a process to change its own session, to change it just once, and to change it to a new session and become the leader. The second issue is preserving thread group relationships, which arises in Linux because of its threading model which treats threads as special processes; this issue does not arise in operating system implementations which do not treat threads as processes. Hereinafter we assume the threading model of Linux 2.6 in which threads are grouped into thread groups with a single thread group leader, which is always the first thread in the group. A thread must be created by its thread group leader because the operating system provides no other way to set the thread group. Given the correct handling of session identifiers and thread groups, other relationships and shared resources can be manipulated after process creation using the operating system interface, and are hence simply assigned once all processes have been created.

Since these two process relationships must be established at process creation time to correctly construct a process forest, the order in which processes are created is crucial. Simply reusing the parent-child relationships maintained by the kernel to create a matching process forest is not sufficient since the forest depends on more than

\(^1\)The term *session* here is unrelated to DejaView’s desktop session; Unix sessions group together process groups, primarily to implement login sessions in textual user interfaces. We refer to this meaning in the remainder of this section.
the process hierarchy at the time of checkpoint. For example, it is important to know
the original parent of a process to ensure that it inherits its correct SID, however
since orphaned children are promptly re-parented to init, the information about
their original parent is lost. While one could log all process creations and deletions
to later determine the original parent, this adds unnecessary runtime overhead and
complexity.

We introduce two algorithms – DumpForest and MakeForest – that use existing
operating system interfaces to efficiently save and restore the process forest, respec-
tively. The algorithms correctly restore a process forest at restart that is the same as
the original process forest at checkpoint. However, they do not require any state other
than what is available at checkpoint time because they do not necessarily recreate
the matching process forest in the same way as the original forest was created.

6.5.1 DumpForest Algorithm

The DumpForest algorithm captures the state of the process forest in two passes. It
runs in linear time with the number of process identifiers in use in a pod. A process
identifier is in use not only if a process exists, but also even if a process has terminated
but as long as the identifier is still being used, for example as an SID for some session
group. The first pass scans the list of processes identifiers within the pod and fills in
a table of entries; the table is not sorted. Each entry in the table represents a PID in
the forest. The second pass records the process relationships by filling in information
in each table entry. A primary goal of this pass is to determine the creating parent
(creator) of each process, including which processes have init as their parent. At
restart, those processes will be created first to serve as the roots of the forest, and
will recursively create the remaining processes as instructed by the table.
Table 6.1 – Possible flags in the status field

<table>
<thead>
<tr>
<th>Flag</th>
<th>Property of Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dead</td>
<td>Corresponds to a terminated process</td>
</tr>
<tr>
<td>Session</td>
<td>Inherits ancestor SID before parent changes its own</td>
</tr>
<tr>
<td>Thread</td>
<td>A thread but not a thread group leader</td>
</tr>
<tr>
<td>Sibling</td>
<td>Created by sibling via parent inheritance</td>
</tr>
</tbody>
</table>

Each entry in the table consists of the following set of fields: status, PID, SID, thread group identifier, and three pointers to the table locations of the entry’s creator, next sibling, and first child processes to be used by MakeForest. Note that these processes may not necessarily correspond to the parent, next sibling, and first child processes of a process at the time of checkpoint. Table 6.1 lists the possible flags for the status field. In particular, Linux allows a process to be created by its sibling, thereby inheriting the same parent, which differs from traditional parent-child only fork creation semantics; a Sibling flag is necessary to note this case.

6.5.1.1 Basic Algorithm

For simplicity, we first assume no parent inheritance in describing the DumpForest algorithm. We divide it into three procedures, namely DumpForest, FindCreator, and AddPlaceHolder, shown in Algorithms 6.1–6.3, respectively.

The first pass of the algorithm initializes the PID and SID fields of each entry according to the process it represents, and all remaining fields to be empty. As shown in Algorithm 6.1, the second pass calls FindCreator on each table entry to populate the empty fields and alter the status field as necessary. The algorithm can be thought of as determining under what circumstances the current parent of a process at time of checkpoint cannot be used to create the process at restart.

The algorithm looks at each table entry and determines what to do based on the properties of the entry. If the entry is a thread and not the thread group leader, we
Algorithm 6.1: DumpForest

1 Procedure DumpForest
2 begin
3   foreach process proc in the forest do
4       Add new entry ent to table;
5       if proc is dead then
6           ent.status |= DEAD;
7       end if
8   end for
9   foreach entry ent in the table do
10      if ent.creator is NIL then
11         FindCreator(ent); /* See Algorithm 6.2 */
12      end if
13   end for
14 end

mark its creator as the thread group leader and add Thread to its status field so that it must be created as a thread on restart. The thread group leader can be handled as a regular process, and hence is treated as part of the other cases.

Otherwise, if the entry is a session leader, this is an entry that at some point called setsid. It does not need to inherit its session from anyone, so its creator can just be set to its parent. If a pod had only one session group, the session leader would be at the root of the forest and its parent would be init.

Otherwise, if the entry corresponds to a dead process (no current process exists with the given PID), the only constraint that must be satisfied is that it inherit the correct session group from its parent. Its creator is just set to be its session leader. The correct session group must be maintained for a process that has already terminated because it may be necessary to have the process create other processes before terminating itself, to ensure that those other processes have their session groups set correctly.
Algorithm 6.2: FindCreator

Procedure FindCreator(ent)
begin
leader ← session leader entry;
if ent is a dead process then
    parent ← init;
    ent.status |= DEAD;
else
    parent ← parent process entry;
end if
if ent is thread (but not thread group leader) then
    ent.creator ← thread group leader;
    ent.status |= THREAD;
else if ent == leader then
    ent.creator ← parent;
else if ent.status & DEAD then
    ent.creator ← leader;
else if parent == init then
    Add-placeholder(ent, leader); /* See Algorithm 6.3 */
else if ent.sid == parent.sid then
    ent.creator ← parent;
else
    ent.creator ← parent;
    sid ← ent.sid;
    repeat
    ent.status |= session;
    if ent.creator == init then
        Add-placeholder(ent, leader); /* See Algorithm 6.3 */
    end if
    ent ← ent.creator;
    if ent.creator is NIL then
        FindCreator(ent);
    end if
    until ent.sid == sid or ent.status & Session;
end if
end

Note: when the creator field is set, the matching child and sibling fields are adjusted accordingly; details are omitted for brevity.
Algorithm 6.3: AddPlaceHolder

1 Procedure AddPlaceHolder(ent, leader)
2 begin
3   Add new entry new to table;
4   new.creator ← leader;
5   new.status = DEAD;
6   ent.creator ← new;
7 end

Note: when the creator field is set, the matching child and sibling fields are adjusted accordingly; details are omitted for brevity.

Otherwise, if the entry corresponds to an orphan process, it cannot inherit the correct session group from init. Therefore, we add a place-holder process in the table whose function on restart is to inherit the session group from the entry’s session leader, create the process, then terminate so that the process will be orphaned. The place-holder is assigned an arbitrary PID that is not already in the table, and the SID identifying the session. To remember to terminate the place-holder process, the place-holder entry’s status field is marked Dead.

Otherwise, if the entry’s SID is equal to its parent’s, the only constraint that must be satisfied is that it inherit the correct session group from its parent. This is simply done by setting its creator to be its parent.

If none of the previous cases apply, then the entry corresponds to a process which is not a session leader, does not have a session group the same as its parent, and therefore whose session group must be inherited from an ancestor further up the process forest. This case arises because the process was forked by its parent before the parent changed its own SID. Its creator is set to be its parent, but we also mark its status field Session, indicating that at restart the parent will need to fork before (potentially) creating a new session. When an entry is marked with Session, it is necessary to propagate this attribute up its ancestry hierarchy until an entry with
that session group is located. In the worst case, this would proceed all the way to the session leader. This is required for the SID to correctly descend via inheritance to the current entry. Note that going up the tree does not increase the runtime complexity of the algorithm because traversal does not proceed beyond entries that already possess the Session attribute.

If the traversal fails to find an entry with the same SID, it will stop at an entry that corresponds to a leader of another session. This entry must have formerly been a descendant of the original session leader. Its creator will have already been set to init. Because we now know that it needs to pass the original SID to its own descendants, we re-parent the entry to become a descendant of the original session leader. This is done using a place-holder in a manner similar to how we handle orphans that are not session leaders.

### 6.5.1.2 Examples

Figure 6.1 illustrates the output of the algorithm on a simple process forest. Figure 6.1a shows the process forest at checkpoint time. Figure 6.1b shows the table generated by DumpForest. The algorithm first creates a table of seven entries corresponding to the seven processes, then proceeds to determine the creator of each entry. Processes 502, 505, and 506 have their Session attributes set, since they must be forked off before their parents’ session identifiers are changed. Note that process 505 received this flag by propagating it up from its child process 506.

Figure 6.2 illustrates the output of the algorithm on a process forest with a missing process, 501, which exited before the checkpoint. Figure 6.2a shows the process forest at checkpoint time. Figure 6.2b shows the table generated by DumpForest. While the algorithm starts with six entries in the table, the resulting table has nine entries since three place-holder processes, 997, 998, and 999, were added to maintain proper
process relationships. Observe that process 503 initially has its creator set to `init`, but is re-parented to the place-holder 998 as part of propagating its child’s \texttt{Session} attribute up the tree.

### 6.5.1.3 Linux Parent Inheritance

We now discuss modifications to the basic DumpForest algorithm for handling the unique parent inheritance feature in Linux which allows a process to create a sibling. To inherit session relationships correctly, parent inheritance must be accounted for in determining the creators of processes that are not session leaders.

If its session leader is alive, we can determine that a process was created by its sibling if its parent is the creator of the session leader. If the session leader is dead, this check will not work since its parent is now `init` and there is no longer any information about its original parent. After a process dies, there is no easy way to determine its original parent in the presence of parent inheritance.
To provide the necessary information, we instead record the session a process inherits when it is created, if and only if the process is created with parent inheritance and it is a sibling of the session group leader. This saved-SID is stored as part of the process’s virtualization data structure so that it can be used later if the process remains alive when the forest needs to be saved. A process created with parent inheritance is a sibling of the session group leader if either its creator is the session group leader, or its creator has the same saved-SID recorded, since the sibling relationship is transitive.

To support parent inheritance, we modify Algorithm 6.2 by inserting a new conditional after the check for whether an entry’s SID is equal to its parent’s and before the final else clause in FindCreator. The conditional examines whether an entry’s saved-SID has been set. If it has been set and there exists another entry in the process forest table whose PID is equal to this saved-SID, the entry’s status field is marked Sibling so that it will be created with parent inheritance on restart. The entry’s
creator is set to the entry that owns that PID, which is leader of the session identified by the saved-SID. Finally, the creator of this session leader is set to be the parent of the current process, possibly re-parenting the entry if its creator had already been set previously.

### 6.5.2 MakeForest Algorithm

Given the process forest data structure, the MakeForest algorithm is straightforward, as shown in Algorithms 6.4–6.6. It reconstructs the process hierarchy and relationships by executing completely in user mode using standard system calls, minimizing dependencies on any particular kernel internal implementation details. The algorithm runs in linear time with the number of entries in the forest. It works in a recursive manner by following the instructions set forth by the process forest data structure. MakeForest begins with a single process that will be used in place of init to fork the processes that have init set as their creator. Each process then creates its own children.

The bulk of the algorithm loops through the list of children of the current process three times during which the children are forked or cleaned up. Each child that is forked executes the same algorithm recursively until all processes have been created. In the first pass through the list of children, the current process spawns children that

---

**Algorithm 6.4: MakeForest**

1. Procedure MakeForest
2. begin
3. foreach entry ent in the table do
4. if ent.creator is init then
5. ForkChildren(ent); /* See Algorithm 6.5 */
6. end if
7. end for
8. end
Algorithm 6.5: ForkChildren

1 Procedure ForkChildren(ent)
2 begin
3 foreach Child cld of ent do
4     if cld.status & SESSION then
5         ForkChild(cld); /* See Algorithm 6.6 */
6     end if
7 end for
8 if ent.sid == ent.pid then
9     setsid();
10 end if
11 foreach child cld of ent do
12     if not cld.status & SESSION) then
13         ForkChild(cld); /* See Algorithm 6.6 */
14     end if
15 end for
16 foreach child cld of ent do
17     if cld.status & DEAD then
18         waitpid(cld.pid);
19     end if
20 end for
21 if ent.status & DEAD then
22     exit(cld.exit);
23 else
24     RestoreProcessState(cld);
25 end if
26 end
Chapter 6. Live Execution Recording

Algorithm 6.6: ForkChild

1 Procedure ForkChild(cld)
2 begin
3 if cld.status & THREAD then
4     pid ← fork_thread();
5 else if cld.status & SIBLING then
6     pid ← fork_sibling();
7 else
8     pid ← fork();
9 end if
10 if pid is 0 then
11     ForkChildren(cld); /* See Algorithm 6.5 */
12 end if
13 end

are marked Session and thereby need to be forked before the current session group is changed. The process then changes its session group if needed. In the second pass, the process forks the remainder of the children. In both passes, a child that is marked Thread is created as a thread and a child that is marked Sibling is created with parent inheritance. In the third pass, terminated processes and temporary placeholders are cleaned up. Finally, the process either terminates if it is marked Dead or calls RestoreProcessState() which does not return. RestoreProcessState() restores the state of the process to the way it was at the time of checkpoint.

6.6 Shared Resources

After the process hierarchy and relationships have been saved or restored, we process operating system resources that may be shared among multiple processes. They are either globally shared at the pod level, such as IPC identifiers and pseudo terminals, or locally shared among a subset of processes, such as virtual memory, file descriptors, signal handlers and so forth. As discussed in § 6.1.1, globally shared resources are
processed first, then locally shared resources are processed. Shared resources may be referenced by more than one process, yet their state need only be saved once. We need to be able to uniquely identify each resource, and to do so in a manner independent of the operating system instance to be able to restart on another instance.

Every shared resource is represented by a matching kernel object whose kernel address provides a unique identifier of that instance within the kernel. We represent each resource by a tuple of the form $\langle \text{Address}, \text{Tag} \rangle$, where $\text{address}$ is the kernel address of the object, and $\text{tag}$ is a serial number that reflects the order in which the resources were encountered (counting from 1 and on). Since tags are mapped one-to-one to unique kernel addresses, they are, therefore, unique logical identifiers for resources. The tuples allow the same resource representation to be used for both checkpoint and restart mechanisms, simplifying the overall implementation. During checkpoint and restart, the tuples are stored in an associative memory in the kernel, enabling fast translation between physical and logical resource identifiers. Tuples are registered into the memory as new resources are discovered, and discarded once the entire checkpoint (or restart) is completed. This memory is used to decide whether a given resource (physical or logical for checkpoint or restart respectively) is a new instance or merely a reference to one already registered. Both globally and locally shared resources are stored using the same associative memory.

During checkpoint, as the processes within the pod are scanned one by one, the resources associated with them are examined by looking up their kernel addresses in the associative memory. If the entry is not found (that is, a new resource detected) we allocate a new tag, register the new tuple and record the state of that resource. The tag is included as part of that state. If the entry is found, it means that the resource is shared and has been already dealt with earlier. Hence it suffices to record its tag for later reference. Note that the order of the scan is insignificant.
During restart, the algorithm restores the state of the processes and the resources they use. The data is read in the same order as has been written originally, ensuring that the first occurrence of each resource is accompanied with its actual recorded state. For each resource identifier, we examine whether the tag is already registered, and if not we create a new instance of the required resource, restore its state from the checkpoint data, and register an appropriate tuple, with the address field set to the kernel address that corresponds to the new instance. If a tuple with the specified tag is found, we locate the corresponding resource with the knowledge of its kernel address as taken from the tuple.

### 6.6.1 Nested Shared Objects

Nested sharing occurs in the kernel when a common resource is referenced by multiple distinct resources rather than by processes. One example are objects that represent a FIFO in the file system, as a FIFO is represented by a single inode which is in turn pointed to by file descriptors of reader and writer ends. Another example is a single backing file that is mapped multiple times within distinct address spaces (or even within the same address space). In both examples shared objects—file descriptors and address spaces respectively—refer to a shared object, yet may themselves be shared by multiple processes.

Nested sharing is handled similarly to simple sharing. To ensure consistency we enforce an additional rule, namely that a nested object is always recorded prior to the objects that point to it. For instance, when saving the state of a file descriptor that points to a FIFO, we first record the state of the FIFO. This ensures that the tuples for the nested resource exist in time for the creation of referring object.
6.6.2 Compound Shared Objects

Many instances of nested objects involve a pair of coupled resources. For example, a single pipe is represented in the kernel by two distinct inodes that are coupled in a special form, and Unix domain sockets can embody up to three disparate inodes for the listening, accepting and connecting sockets. We call such objects *compound objects*. Unlike unrelated resources, compound objects have two or more internal elements that are created and interlinked with the invocation of the appropriate kernel subroutine(s) such that their lifespans are correlated, e.g., the two inodes that constitute a pipe.

We consistently track a compound object by capturing the state of the entire resource including all components at once, at the time it is first detected, regardless of through which component it was referred. On restart, the compound object will be encountered for the first time through some component, and will be reconstructed in its entirety, including all other components. Then only the triggering component (the one that was encountered) will need to be attached to the process that owns it. The remaining components will linger unattached until they are claimed by their respective owners at a later time.

The internal ordering of the elements that compose a compound object may depend on the type of the object. If the object is symmetric, such as *socketpairs*, its contents may be saved at an arbitrary order. Otherwise, the contents are saved in a certain order that is particularly designed to facilitate the reconstruction of the object during restart. For example, the order for pipes is first the read-side followed by the write-side. The order for Unix domain sockets begins with the listening socket (if it exists), followed by the connecting socket and finally the accepting socket. This order reflects the sequence of actions that is required to rebuild such socket-trios:
first create a listening socket, then a socket that connects to it, and finally the third socket by accepting the connection.

### 6.6.3 Memory Sharing

Since memory footprint is typically the most dominant factor in determining the size of the checkpoint image, we now further discuss how recording shared resources is done in the case of memory. A memory region in the address space of a process can be classified along two dimensions, one is whether it is mapped to a backing file or anonymous, and the other is whether it is private to some address space or shared among multiple ones. For example, text segments such as program code and shared libraries are mapped and shared, IPC shared memory is anonymous and shared, the data section is mapped and private, and the heap and stack are anonymous and private.

Memory sharing can occur in any of these four cases. Handling regions that are shared is straightforward. If a region is mapped and shared, it does not need to be saved since its contents are already on the backing file, and any buffered data is written as part of the file system snapshot. If a region is anonymous and shared, it is treated as a normal shared object so that its contents are only saved once. Handling regions that are private is more subtle. While it appears contradictory to have memory sharing with private memory regions, sharing occurs due to the kernel’s COW optimization. When a process forks, the kernel defers the creation of a separate copy of the pages for the newly created process until one of the processes sharing the common memory attempts to modify it. During checkpoint, each page that has been previously modified and belongs to a private region that is marked COW is treated as a nested shared object so that its contents are only saved once. During restart, the
COW sharing is restored. Modified pages in either anonymous and private regions or mapped and private regions are treated in this manner.

6.7 Evaluation

To demonstrate the effectiveness of our approach, we have implemented a checkpoint-restart prototype as a Linux kernel module and associated user-level tools and evaluated its performance on a wide range of real applications. We also quantitatively compared our prototype with two other commercial virtualization systems, OpenVZ and Xen. OpenVZ provides another operating system virtualization approach for comparison, while Xen provides a hardware virtualization approach for comparison. We used the latest versions available at the time of our experiments for both.

The measurements were conducted on an IBM HS20 eServer BladeCenter, each blade with dual 3.06 GHz Intel Xeon™ CPUs, 2.5 GB RAM, a 40 GB local disk, and Q-Logic Fibre Channel 2312 host bus adapters. The blades were interconnected with a Gigabit Ethernet switch and linked through Fibre Channel to an IBM FastT500 SAN controller with an Exp500 storage unit with ten 70 GB IBM Fibre Channel hard drives. Each blade used the GFS cluster file system [155] to access a shared SAN. Unless otherwise indicated, the blades were running Debian 3.1 distribution and the Linux 2.6.11.12 kernel.

Table 6.2 lists the nine application scenarios used for our experiments. The scenarios were running an Apache web server, a kernel compile, a MySQL database server, a volano chat server, an entire operating system at user-level using UML (User Mode Linux [46]), and four desktop applications scenarios run using a full Gnome X desktop environment with an XFree86 4.3.0.1 server and THINC [15] to provide remote display access to the desktop. The four desktop scenarios were running a baseline
environment without additional applications, a web browser, a video player, and a Microsoft Office suite using CrossOver Office. The UML scenario shows the ability to checkpoint and restart an entire operating system instance. The Microsoft Office scenario shows the ability to checkpoint and restart Windows applications using CrossOver Office on Linux.

We measured checkpoint-restart performance by running each of the application scenarios and taking a series of ten full (i.e., non-incremental) checkpoints during their execution. We omitted file system snapshots to focus on the performance of saving and restoring only the execution state. We measured the checkpoint image sizes, number of processes that were checkpointed, checkpoint times, and restart times, then averaged the measurements across the ten checkpoints for each application scenario. Figures 6.3 to 6.8 show results for our checkpoint-restart prototype.

Figure 6.3 shows the average total checkpoint image size, as well as a breakdown showing the amount of data in the checkpoint image attributable to the process forest. The total amount of state that is saved is modest in each case and varies according to the applications executed, ranging from a few MBs on most applications to tens
Figure 6.3 – Average checkpoint size

of MBs for graphical desktop sessions. The results show that the total memory in use within the pod is the most prominent component of the checkpoint image size, accounting for over 99% of the image size.

An interesting case is UML, which uses memory mapping to store guest main memory using an unlinked backing file. This file is separate from memory and amounts to 129 MB. By using the optimization for unlinked files as discussed in §6.1.1 and storing the unlinked files separately on the file system, the UML state stored in the checkpoint image can be reduced to roughly 1 MB. The same occurs for CrossOver Office, which also maps additional 16 MB of memory to an unlinked backing file.

Figure 6.4 shows the average number of processes running within the pod at checkpoints for each application scenario. On average the process forest tracks 35 processes in most scenarios, except for apache and volano with 169 and 839 processes each, most of which are threads. As Figure 6.3 shows the process forest always occupies a small fraction of the checkpoint, even for volano.
Chapter 6. Live Execution Recording

Figure 6.4 – Average number of processes

Figure 6.5 – Average checkpoint time
Figure 6.5 shows the average total checkpoint times for each application scenario, which is measured from when the pod is quiesced until the complete checkpoint image is written out to disk. We also show two other measures. Checkpoint downtime is the time from when the pod is quiesced until the pod can be resumed; it is the time to record the checkpoint data without committing it to disk. Sync checkpoint time is the total checkpoint time plus the time to force flushing the data to disk. Average total checkpoint times are under 600 ms for all application scenarios and as small as 40 ms, which is the case for UML. Comparing with Figure 6.3, the results show that both the total checkpoint times and the sync times are strongly correlated with the checkpoint sizes. Writing the file system, particularly with forced flushing of the data to disk, is largely limited by the disk I/O rate. For example, gnome-base has an average checkpoint size of 39 MB and an average sync checkpoint time of just under 3 s. This correlates directly with the sustained write rate for GFS, which was roughly 15 MB/s in our measurements.

Perhaps more importantly, checkpoint downtimes in Figure 6.5 show that the average time to actually perform the checkpoint without incurring storage I/O costs is small, ranging from 12 ms for a kernel make to at most 90 ms for a full fledged desktop running Microsoft Office. Though an application is unresponsive while it is quiesced and being checkpointed, even the largest average checkpoint downtimes are less then 100 ms. Furthermore, the average checkpoint downtimes were less than 50 ms for all application scenarios except Microsoft Office.

Figure 6.6 compares the checkpoint downtime for each application scenario with and without the memory buffering and COW mechanisms that we employ. Without these optimizations, checkpoint data must be written out to disk before the pod can be resumed, resulting in checkpoint downtimes that are close to the total checkpoint times shown in Figure 6.5. The memory buffering and COW checkpoint optimization
reduce downtime from hundreds of milliseconds to almost always under 50 ms, in some cases even as much as an order of magnitude.

Figure 6.7 shows the breakdown of the total checkpoint time (excluding sync) for each application scenario, as percentage of the total time attributable to different steps: *quiesce*—the time to quiesce the pod, *record*—the time to record the checkpoint data, and *commit*—the time to commit the data by writing it out to storage. The commit step amounts to 80-95% of the total time in almost all application scenarios, except for UML where it amounts to only 15% due to a much smaller checkpoint size. Quiescing the processes took less then 700 µs for all application scenarios except *apache* and *volano*, which took roughly 1.5 ms and 5 ms, respectively. The longer quiesce times are due to the large number of processes being executed in *apache* and *volano*. The time to generate and record the process forest was even smaller, less than 10 µs for all applications except *apache* and *volano*, which took 30 µs and 336 µs respectively. The longer times were mainly due to allocation of extra memory. This
could be improved by estimating the number of entries in the structure using the actual number of processes in the pod to preallocate sufficient pages to hold that many entries. The time to record globally shared resources was also quite small, under 10 µs in all cases.

Figure 6.8 presents the average total restart times for each application scenario. The restart times were measured for two distinct configurations: *warm cache*—restart was done with a warm file system cache immediately after the checkpoint was taken, *cold-cache*—restart was done with a cold file system cache after the system was rebooted, forcing the system to read the image from the disk. Warm cache restart times were less than .5 s in all cases, ranging from 24 ms for *apache* to 386 ms for a complete Gnome desktop running Microsoft Office. Cold cache restart times were longer as restart becomes limited by the disk I/O rate. Cold cache restart times were less than 2 s in all cases, ranging from 65 ms for UML to 1.9 s for Microsoft Office. The cold restart from a checkpoint image is still noticeably faster than the checkpoint
Figure 6.8 – Average restart time

to the file system with flushing because GFS file system read performance is much faster than its write performance.

To provide a comparison with another operating system virtualization approach, we also performed our experiments with OpenVZ. We used version 2.6.18.028stab on the same Linux installation. Because of its lack of GFS support, we copied the installation to the local disk to conduct experiments. Since this configuration is different from what we used with our prototype, the measurements are not directly comparable. However, they provide some useful comparisons between the two approaches. We report OpenVZ results for apache, make, mysql and volano; OpenVZ was unable to checkpoint the other scenarios. Table 6.3 presents the average total checkpoint times, warm cache restart times, and checkpoint image sizes for these applications. We ignore sync checkpoint times and cold cache restart times to reduce the impact of the different disk configurations used.
Table 6.3 – Checkpoint-restart performance for applications that worked on OpenVZ

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Checkpoint [s]</th>
<th>Restart [s]</th>
<th>Size [MB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>0.730</td>
<td>1.321</td>
<td>7.7</td>
</tr>
<tr>
<td>make</td>
<td>2.230</td>
<td>1.376</td>
<td>53</td>
</tr>
<tr>
<td>mysql</td>
<td>1.793</td>
<td>1.288</td>
<td>22</td>
</tr>
<tr>
<td>volano</td>
<td>2.036</td>
<td>1.300</td>
<td>25</td>
</tr>
</tbody>
</table>

The results show that OpenVZ checkpoint and restart times are significantly worse than our system. OpenVZ checkpoint times were 5.2, 5.6, 12.4, and 3.0 times slower for apache, make, mysql and volano, respectively. OpenVZ restart times were 55.0, 6.6, 29.9, and 5.0 times slower for apache, make, mysql and volano, respectively. OpenVZ checkpoint sizes were .48, 1.3, 1.2, and .46 times the sizes of our system. The difference in checkpoint sizes was relatively small and does not account for the huge difference in checkpoint-restart times even though different file system configurations were used due to OpenVZ’s lack of support for GFS. OpenVZ restart times did not vary much among application scenarios, suggesting that container setup time may constitute a major component of latency.

To provide a comparison with a hardware virtualization approach, we performed our experiments with Xen. We used Xen 3.0.3 with its default Linux 2.6.16.29. We were unable to find a GFS version that matched this configuration, so we used the local disk to conduct experiments. We also used Xen 2.0 with Linux 2.6.11 because this configuration worked with GFS. In both cases, we used the same kernel for both “dom0” and “domU”. We used three VM configurations with 128 MB, 256 MB, and 512 MB of memory. We report results for apache, make, mysql, UML, and volano; Xen was unable to run the other scenarios due to lack of support for virtual consoles. Table 6.4 presents the average total checkpoint times, warm cache restart times, and checkpoint image sizes for these applications. We report a single number for each
configuration instead of per application since Xen results were directly correlated with the VM memory configuration and did not depend on the applications scenario. Checkpoint image size was determined by the amount of RAM configured. Checkpoint and restart times were directly correlated with the size of the checkpoint images.

The results show that Xen checkpoint and restart times are significantly worse than our system. Xen 3 checkpoint times were 5.2 (volano on 128 MB) to 563 (UML on 512 MB) times slower. Xen 3 restart times were 6.2 (volano on 128 MB) to 1137 (apache on 512 MB) slower. Xen results are also worse than OpenVZ; both operating system virtualization approaches performed better. Restart times for the 256 MB and 512 MB VM configurations were much worse than the 128 MB VM because the Xen checkpoint images ended up being too large to be effectively cached in the kernel, severely degrading warm cache restart performance to roughly the same level as with a cold cache. Note that although precopying can reduce application downtime for Xen migration [35], it will not reduce total checkpoint-restart times, which can be two orders of magnitude slower.

### Table 6.4 – Checkpoint-restart performance for Xen VMs

<table>
<thead>
<tr>
<th>Xen Config.</th>
<th>Checkpoint [s]</th>
<th>Restart [s]</th>
<th>Image Size [MB]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xen 3</td>
<td>Xen 2</td>
<td>Xen 3</td>
</tr>
<tr>
<td>128 MB</td>
<td>3.5</td>
<td>5.5</td>
<td>1.6</td>
</tr>
<tr>
<td>256 MB</td>
<td>10.3</td>
<td>12</td>
<td>13.4</td>
</tr>
<tr>
<td>512 MB</td>
<td>25.9</td>
<td>19</td>
<td>27.3</td>
</tr>
</tbody>
</table>

6.8 Summary

In this chapter, we presented DejaView’s architecture for recording and later reviving live execution. We introduced a transparent mechanism for checkpoint-restart in
commodity operating systems that can checkpoint multiple processes in a consistent
manner so that they can be restarted correctly at a later time. Our approach com-
bines a kernel-level checkpoint mechanism with a hybrid user-level and kernel-level
restart mechanism to leverage existing operating system interfaces and functional-
ity. We introduced novel algorithms for saving and restoring process relationships
and for efficient handling of shared state across cooperating processes. We presented
checkpoint optimizations for recording application execution state without affecting
interactive desktop performance. We also presented a coordinated checkpoint and
file system mechanism that combines log structured and unioning file systems to en-
able fast file system snapshots consistent with checkpoints, allowing checkpoints to
be later revived for simultaneous read-write usage. Our experimental results with a
checkpoint-restart prototype show that DejaView’s live execution recording generates
modest checkpoint image sizes and provides fast checkpoint and restart times without
modifying existing system components. Comparisons with two commercial systems
demonstrate that our prototype provides much faster checkpoint-restart performance
and more robust checkpoint-restart functionality than these other approaches.
Chapter 7

Whole System Evaluation

In the previous Chapter 4, Chapter 5, and Chapter 6, we have discussed and evaluated three components used as building blocks for DejaView, namely DejaView’s display-centric text recorder, virtual execution environment, and live execution checkpoint-restart mechanism. Through the evaluations we have quantified the individual components demonstrating that they meet the desired functionality and performance requirements. In this chapter, we combine them all together to provide a whole system evaluation of DejaView.

We have implemented a DejaView prototype 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 userspace 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 [91] and UnionFS [183] that guarantees consistency between checkpoints and file system state. For capturing text information, DejaView uses the accessibility infrastructure of the GNOME desktop environment [1]. Indexing and searching text is performed using the Tsearch extension [167] 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.

We present experimental data that quantifies the performance of DejaView when running a variety of common desktop applications using this prototype. We present results for both application benchmarks and real user desktop usage. We focus on evaluating the performance of using DejaView along two dimensions, namely overall system overhead and usability in terms of latency to access recorded data. We quantify the recording runtime overhead, the impact on system interactivity, and the storage requirements of using DejaView in terms of the cost of continuously recording display and execution. We quantify the text search latency, the display browse latency and playback speed, and the session revive latency, to show DejaView’s usability in terms of providing efficient access to recorded content.

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 checkpointed according to the policy described in §6.2.3 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 virtualization overhead results since we have already shown in §5.4 that our virtual execution environments impose low overhead, and previous work has shown that our basic virtual display mechanism provides good display performance [15].

We used the desktop application scenarios listed in Table 7.1. 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
### Table 7.1 – Application scenarios

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

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.
7.1 System Overhead

Figure 7.1 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 [119]. 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 7.1 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 7.1 additionally shows the content recording overhead, which is small in all scenarios except for the web benchmark. The overhead is less than 4% for all cases except for the web benchmark. For the web benchmark, the content recording 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 the accessibility features of Firefox improve [57].

Figure 7.2 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 *write-back*, 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 7.2 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 [150]. 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 7.3 – Recording storage growth

Figure 7.2 shows that pre-checkpoint and write-back 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 write-back 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 7.3 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 7.3 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 a growth rate, resulting in much lower storage requirements in practice. As shown in Figure 7.3, the storage space growth rate for real user desktop usage is much more modest at only 2.5 MB/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 [127], 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.

7.2 Access To Data

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 7.4 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 dissertation.

We measured browsing performance (time to access and display specific contents from the display recording) 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
Figure 7.5 – Playback speedup

used, and hence are unlikely to be of interest to the user. Figure 7.4 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 7.5 demonstrates that DejaView is able to playback an entire record at many times the rate at which it was originally generated. For instance, Figure 7.5 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 7.6 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 can occur when users revive a session at a time relatively close to the current time. Perhaps a more common scenario of reusing cached checkpoint files is when a user conducts a specific search and needs to revive her session from multiple points in time that are near each
other. While these points in time represent distinct session states, they are likely to reference a common set of incremental checkpoints.

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.

7.3 Summary

In this chapter, we presented a whole system experimental evaluation of a prototype implementation of DejaView on common desktop application workloads and with real desktop usage. The prototype is implemented as a set of loadable modules for Linux and the X Window System and provides transparent operation without modifying existing system components. Our results show that DejaView’s recording of display and execution state adds negligible overhead of only a few milliseconds of interruption to interactive applications, which is typically not noticeable to end users even for more time-sensitive applications such as movie playback. DejaView’s playback can enable users to quickly view display records at up to 270 times faster than real-time, and browsing and searching display information is fast enough to be done at interactive rates. DejaView’s storage requirements at highest quality are comparable to PVRs in recording HDTV resolution media programming, enabling high quality recording for everyday use as terabyte storage capacities become commonplace. These results demonstrate the architecture’s effectiveness in providing continuous low-overhead recording with unnoticeable overhead.
Many of the benefits of DejaView’s checkpoint-restart mechanism from Chapter 6 become crucial in cluster computing environments, including fault resilience and recovery, improved resources utilization and load balancing, and improved maintenance and administration. These benefits are particularly salient for long-running jobs where the ability to restart a job in the middle of its execution instead of needing to start over from the beginning is important. Recognizing the importance of providing these capabilities for clusters, in this chapter we extend DejaView’s checkpoint-restart mechanism for distributed applications that run on multiple nodes in a cluster.

For distributed applications, a checkpoint-restart mechanism needs to not only save and restore the application state associated with each cluster node, but it must also ensure that the state saved and restored across all participating nodes is globally consistent. Checkpoint and restart must be coordinated across all participating nodes to ensure that application processes running on each node are synchronized correctly. In particular, the network state of communication links among application processes on different nodes must be checkpointed and restarted such that nodes properly agree on the state of messages being delivered. If a node’s state reflects a message receipt,
then the state of the corresponding sender should reflect having sent that message [29]. Although coordinated checkpoint-restart of distributed applications provides substantial potential benefits, existing approaches [7, 23, 28, 30, 54, 140, 146, 163] have been unable to provide this functionality transparently on clusters running commodity operating systems and hardware.

We present ZapC, an extension of DejaView’s checkpoint-restart mechanism that provides transparent coordinated checkpoint-restart of distributed network applications on commodity clusters. ZapC can checkpoint an entire distributed application across all nodes in a coordinated manner so that it can be restarted from the checkpoint on a different set of cluster nodes at a later time. In checkpointing and restarting a distributed application, ZapC separates the processing of network state from per node application state. It only requires synchronized operation in capturing the network state, which represents a small fraction of the overall checkpoint time. Per node checkpoint-restart operations of (non-network) application state proceed in parallel with minimal synchronization requirements among nodes, resulting in faster checkpoint and restart times. ZapC can also directly stream checkpoint data from one set of nodes to another, enabling direct migration of a distributed application to a new set of nodes without saving and restoring state from secondary storage.

ZapC uniquely supports complete checkpoint-restart of network state in a transport protocol independent manner without application or library support. It leverages the socket abstraction and correctly saves and restores all socket state, including socket parameters, socket data queues, and minimal protocol specific state. ZapC accomplishes this in a portable manner using the standard socket interface without detailed knowledge of the underlying network protocol data structures. ZapC accounts for network state in a protocol independent manner for reliable and unreliable network protocols, including TCP, UDP and raw IP.
8.1 Architecture Overview

ZapC is designed to checkpoint-restart an entire distributed network application running on a set of cluster nodes. It can be thought of in terms of three logical components: a standalone checkpoint-restart mechanism based on Zap that saves and restores non-network per-node application state, a manager that coordinates a set of agents each using the standalone checkpoint-restart mechanism to save and restore a distributed application across a set of cluster nodes in a consistent manner, and a network checkpoint-restart mechanism that saves and restores all the necessary network state to enable the application processes running on different nodes to communicate. For simplicity, we describe these ZapC components assuming a commodity cluster in which the cluster nodes are running independent commodity operating system instances and the nodes all have access to a shared storage infrastructure. For example, a common configuration would be a set of blade servers or rack-mounted 1U servers running standard Linux and connected to a common SAN or a NAS storage infrastructure.

To execute a distributed application across a set of cluster nodes, ZapC encapsulates the application processes running on each node in a pod to decouple those processes from the underlying host. Recall that pods provide a self-contained unit that can be isolated from the system, checkpointed to secondary storage, migrated to another machine, and transparently restarted. As a pod migrates from one node to another, virtual resources are remapped to real operating system resources. In particular, ZapC only allows applications in pods to see virtual network addresses which are transparently remapped to underlying real network addresses as a pod migrates among different machines. This enables ZapC to migrate distributed applications to any cluster regardless of its IP subnet or addresses.
With ZapC, a distributed application is executed in a manner that is analogous to a regular cluster, ideally placing each application endpoint in a separate pod. For example, on multiprocessor nodes that run multiple application endpoints, each endpoint can be encapsulated in a separate pod. To leverage mobility, it is advantageous to divide the application into many independent pods, since the pod is the minimal unit of migration. This allows for maximum flexibility when migrating the application. ZapC can migrate a distributed application running on $N$ cluster nodes to run on $M$ cluster nodes, where generally $N \neq M$. For instance, a dual-CPU node may host two application endpoints encapsulated in two separate pods. Each pod can thereafter be relocated to a distinct node; they do not need to be migrated together to the same node.

8.2 Distributed Checkpoint-Restart

To checkpoint-restart a distributed network application, ZapC provides a coordinated checkpoint-restart algorithm that uses the pod checkpoint-restart mechanism and a novel network state checkpoint-restart mechanism described in § 8.3. We assume that all the network connections are internal among the participating nodes that compose the distributed application; connections going outside of the cluster are beyond the scope of this paper. Although ZapC allows multiple pods to execute concurrently on the same node, for simplicity, we describe ZapC operation below assuming one pod per node.

Our coordinated checkpointing scheme consists of a Manager client that orchestrates the operation and a set of Agents, one on each node. The Manager is the front-end client invoked by the user and can be run from anywhere, inside or outside the cluster. It accepts a user’s checkpoint or restart request and translates it into a
set of commands to the Agents. The Agents receive these commands and carry them out on their local nodes.

The Manager maintains reliable network connections with the Agents throughout the entire operation. Therefore an Agent failure will be readily detected by the Manager as soon as the connection becomes broken. Similarly a failure of the Manager itself will be noted by the Agents. In both cases, the operation will be gracefully aborted, and the application will resume its execution.

A checkpoint is initiated by invoking the Manager with a list of tuples of the form \(<\text{node}, \text{pod}, \text{URI}>\). This list specifies the nodes and the pods that compose the distributed application, as well as the destination for the checkpointed data (URI). The destination can be either a file name or a network address of a receiving Agent. This facilitates direct migration of an application from one set of nodes to another without requiring that the checkpoint data first be written to some intermediary storage.

The Manager and the Agents execute the checkpoint algorithms given in Algorithms 8.1 and 8.2 respectively. Given a request for a checkpoint, the Manager begins with broadcasting a \texttt{checkpoint} command to all participating nodes. Upon receiving the command, each Agent initiates the local checkpoint procedure, that is divided into four steps: suspending the designated pod, invoking the network-state check-

\begin{verbatim}
Algorithm 8.1: Coordinated Checkpoint (Manager)
1 forall agents do
2 SendCommand(agent, 'checkpoint')
3 forall agents do
4 RecvData(agent, 'meta-data')
5 forall agents do
6 SendCommand(agent, 'continue')
7 forall agents do
8 RecvData(agent, 'status')
\end{verbatim}
point, proceeding with the standalone pod checkpoint, and finalizing the checkpoint. The Agent also performs three companion steps, lines 3, 4–7, and 9–12 in Figure 8.2, which are not directly related to the local checkpoint procedure, but rather to its interaction with the Manager. Lines 4 and 7 both test the same condition, ensuring that the Agent only finishes after having satisfied two conditions: it has reported its status to the Manager, and it received the continue message from the Manager.

Each Agent first suspends its respective pod by sending a SIGSTOP signal to all the processes in the pod to prevent those processes from being altered during checkpoint. To prevent the network state from changing, the Agent disables all network activity going to and from the pod. This is done by leveraging a standard network filtering service to block the links listed in the table; Netfilter [118] comes standard with Linux and provides this functionality. The Agent then obtains the network meta-data of the node, a table of \( \langle \text{state, source, target} \rangle \) tuples showing all network connections of the pod. This is the first information saved by the Agent as part of the checkpoint and is used by the restart procedure to correctly reconstruct the network state. The source and target fields describe the connection endpoint IP addresses

---

**Algorithm 8.2:** Coordinated Checkpoint (Agent)

```
1 BlockNetwork(pod)
2 Checkpoint(pod, network) /* network checkpoint */
3 SendData(manager, 'meta-data')
4 if 'continue' arrived then /* ok to unblock network? */
5     RecvCommand(manager, 'continue')
6     UnblockNetwork(pod)
7 end
8 Checkpoint(pod, standalone) /* standalone checkpoint */
9 if 'continue' not arrived then /* wait for 'continue'? */
10   RecvCommand(manager, 'continue')
11   UnblockNetwork(pod)
12 end
13 SendData(manager, 'status')
```
and port numbers. The state field reflects the state of the connection, which may be full-duplex, half-duplex, closed (in which case there may still be unread data), or connecting. The first three states are for established connections while the last state is a transient state for a not yet fully established connection.

Once the pod’s network is frozen, the Agent checkpoints the network state. When finished, the Agent notifies the Manager that it has concluded its network state checkpoint, and reports its meta-data. It then proceeds to perform the standalone pod checkpoint. The Agent cannot complete the standalone pod checkpoint until the Manager has received the meta-data from all participating Agents, at which point the Manager tells the Agents they can continue. ZapC checkpoints the network state before the other pod state to enable more concurrent checkpoint operation by overlapping the standalone pod checkpoint time with the time it takes for the Manager to receive the meta-data from all participating Agents and indicate that they can continue.

In the last step, the action taken by the Agent depends on the context of the checkpoint. If the application should continue to run on the same node after the checkpoint (i.e., taking a snapshot), the pod is allowed to resume execution by sending a SIGCONT to all the processes. However, should the application processes migrate to another location, the Agent will destroy the pod locally and create a new one at the destination site. In both cases, a file-system snapshot (if desired) may be taken immediately prior to reactivating the pod.

To provide a better understanding of the checkpoint timing requirements, Figure 8.1 illustrates a typical checkpoint timeline. The timeline is labeled with numbers that correspond to the steps of the checkpoint algorithm as described in Figure 8.1. The timeline shows that the entire checkpoint procedure executes concurrently in an asynchronous manner on all participating nodes for nearly its entire
duration. Figure 8.1 shows that the only synchronization point is the “sync” point at the Manager after step 2 and during step 3.

This single synchronization is necessary and sufficient for the checkpoint procedure to be coherent and correct. It is necessary for the Agents to synchronize at the Manager before completing their standalone pod checkpoints and unblocking their networks. Otherwise it would be possible for one node to resume operation, re-engage in network activity, and deliver data to another node that had not begun its checkpoint. This would result in an inconsistent global state, as the state of the latter node will contain data that is not marked as sent in the already-saved state of the former.
The single synchronization is sufficient since every pod ensures consistency by blocking its connections independently of other pods. Once a pod has blocked its connections, there is no interaction with any other pod even if the network of other pods is not yet blocked. The pod is already isolated and does not need to wait for all other pods to block their connections. By not having to wait for other pods initially, the network activity is only blocked for the minimal required time.

A restart is initiated by invoking the Manager with a list of tuples of the form \((\text{node}, \text{pod}, \text{URI})\). This list describes the mapping of the application to nodes and pods, where \text{URI} indicates the location of the checkpoint data. A key requirement of the restart is to restore the network connections of the distributed application. A naive approach would be to manually create the internal kernel data structures and crowd them with the relevant data, but this is not easy and requires intimate knowledge of the protocol implementation, tight cooperation between the peers, and careful adjustments of protocol-dependent parameters. Since ZapC is restarting the entire distributed application, it controls both ends of each network connection. This makes it straightforward to reconstruct the communicating sockets on both sides of each connection using a pair of \text{connect} and \text{accept} system calls. This leverages the standard socket interface for creating network connections and results in a robust, easy to implement and highly portable approach.

Using this approach, the Manager and the Agents execute the restart algorithms given in Algorithms 8.3 and 8.4, which are similar to the checkpoint counterparts. Given a restart request, the Manager begins sending a \text{restart} command to all the Agents accompanied by a modified version of the \text{meta-data}. The \text{meta-data} is used to derive a new network connectivity map by substituting the destination network addresses in place of the original addresses. This will outline the desired mapping of the application to nodes/pods pairs. In the case of a restart on the same set of
Algorithm 8.3: Coordinated Restart (Manager)

forall agents do
1. SendCommand(agent, 'restart')
2. SendData(agent, 'meta-data')
forall agents do
3. RecvData(agent, 'status')

Algorithm 8.4: Coordinated Restart (Agent)

1. CreatePod(pod)
2. RecvData(manager, 'meta-data')
3. Restart(pod, network)  /* restore network state */
4. Restart(pod, standalone)  /* restore standalone state */
5. SendData(manager, 'status')

Nodes (e.g., recovering from a crash), the mapping is likely to remain unmodified. In the case of migration, the mapping will reflect the settings of the alternate execution environment, particularly the network addresses at the target cluster.

As part of the modified meta-data, the Manager provides a schedule that indicates for each connection which peer will initiate and which peer will accept. This is done by tagging each entry as either a connect or accept type. This is normally determined arbitrarily, except when multiple connections share the same source port number. Source port numbers can be set by the application if not already taken or assigned automatically by the kernel; specifically when a TCP connection is accepted, it inherits the source port number from the “listening” socket. To correctly preserve the source port number when shared by multiple connections, these connections must be created in a manner that resembles their original creation, as determined by the above schedule.

The Agents respond to the Manager’s commands by creating an empty pod into which the application will be restored. It then engages the local restart procedure, which consists of three steps: recovering the network connectivity, restoring the net-
work state, and executing the application standalone restart. Once completed, the pod will be allowed to resume execution without further delay.

The recovery of the network connectivity is performed in user space and is fairly straightforward. The *meta-data* that the Agent received from the Manager completely describes the connectivity of the pod, and can be effectively used as a set of instructions to re-establish the desired connections. The Agent simply loops over all the entries (each of type *connect* or *accept*), and performs the suitable action. If the *state* field is other than full-duplex, the status of the connection is adjusted accordingly. For example, a closed connection would have the *shutdown* system call executed after the rest of its state has been recovered.

Generally, these connections cannot be executed in any arbitrary order, or a deadlock may occur. Consider for instance an application connected in a ring topology (each node has two connections - one at each side): a deadlock occurs if every node first attempts to accept a connection from the next node. To prevent such deadlocks, rather than using sophisticated methods to create a deadlock-free schedule, we simply divide the work between two threads of execution. One thread handles requests for incoming connections, and the other establishes connections to remote pods. Hence, there is no specific order at which connections requests should arrive at the Agent. The result is a simple and efficient connectivity recovery scheme, which is trivial to implement in a portable way.

Once the network connectivity has been re-established, the Agent initiates the restart of the network-state. This ensures that we reinstate the exact previous state of all network connections, namely connection status, receive queue, send queue and protocol specific state. Similarly to the distributed checkpoint, the motivation for this order of actions is to avoid forced synchronization points between the nodes at later stages. In turn, this prevents unnecessary idle time, and increases concurrency.
by hiding associated latencies. With this framework, the only synchronization that is required is indirect and is induced by the creation of network connections. As demonstrated in § 8.4, the standalone restore time greatly dominates the total restore time, and fluctuates considerably. Positioning it as the first to execute may lead to imbalances and wasted idle time due to the synchronization that follows. Instead, our scheme manages to both minimize the loss by doing it early, and enable the pods to continue their execution as soon as they conclude their standalone restart.

A key observation about our restart scheme is that it does not require that the network be disabled for any intermediate period. Recall that with checkpoint, the network was shut off to ensure a consistent state. The challenge was to capture the state of live connections that already carry data in the queues, and are likely to be transient. Conversely the re-established network connections are entirely controlled by out restart code. It is guaranteed that no data, but that which we choose to explicitly send, will be transmitted through the connection, until the application resumes execution (which will only occur at the end of the restart).

The final notch of the procedure is the standalone restart, invoked locally by each Agent after the network state has been successfully restored. To conclude the entire operation, each Agent sends a summary message to the Manager, specifying the completion status (failure or success) and the name of the new pod that has been created. The Manager collects this data from all the Agents and reports it back to the user.

8.3 Network State Checkpoint-Restart

The network-state of an application is defined by the collection of the network-states of its communication endpoints. From the application’s standing point, the primary
abstraction of a communication endpoint is a *socket*. A socket is associated with a network protocol upon creation. The application can bind a socket to an address, connect to an address, accept a connection, as well as exchange data. The operating system in turn, keeps a certain amount of state for each socket. The network-state checkpoint-restart is responsible for capturing and restoring this state.

The state of a socket has three components: socket parameters, socket data queues and protocol specific state. The socket parameters describe socket properties related to its state, e.g., connected or not, and to its behavior, e.g., blocking or non-blocking I/O. The data queues, specifically send and receive queues, hold incoming and outgoing data respectively, which is handled by the network layer. Protocol specific data describes internal state held by the protocol itself. For instance, TCP connection state and TCP timers are part of its state.

Saving the state of the socket parameters is fairly straightforward. Recall that while taking a network-state checkpoint, the processes in the pod are suspended and cannot alter the socket state. Also, the network is blocked and is only restarted later on after all the applications involved in the checkpoint have terminated their local network-state checkpoint. That given, the socket parameters can be safely extracted at this point. Furthermore, these properties are user-accessible via a standard interface provided by the operating system, namely `getsockopt` and `setsockopt` system calls. We build on this interface to save the socket parameters during checkpoint and restore it during restart. For correctness, the entire set of the parameters is included in the saved state (for a comprehensive list of such options, refer to [160]).

The socket’s receive and send queues are stored in the kernel. They hold intermediate data that has been received by the network layer but not yet delivered to (read by) the application, as well as data issued by the application that has not yet been transmitted over the network.
With unreliable protocols, it is normally not required to save the state of the queue. Packet loss is an expected behavior and should be accounted for by the application: a specific segment of data not restored can be interpreted as a legitimate packet loss. One exception, however, is if the application has already “peeked” at (that is, examined but not consumed) the receive queue. This is a standard feature in most operating system and is regularly used. To preserve the expected semantics, the data in the queue must be restored upon restart, since its existence is already part of the application’s state. With reliable protocols, on the other hand, the queues are clearly an integral part of the socket state and cannot be dispensed of. Consequently we chose to have our scheme always save the data in the queues, regardless of the protocol in question. The advantage is that it prevents causing artificial packets loss that would otherwise slowdown the application shortly after its restart, the amount of time it lingers until it detects the loss and fixes it by retransmission.

In both cases (reliable and unreliable protocols) in-flight data can be safely ignored. Such data will either be dropped (for incoming packets) or blocked (for outgoing packets) by the network layer, since the pod’s network is blocked for the duration of the checkpoint. With unreliable protocols this is obviously an expected behavior. Reliable protocols will eventually detect the loss of the data and consequently retransmit it.

Saving the state of the receive queue and the send queue necessitates a method to obtain their contents. It is critical that the method be transparent and not entail any side-effects that may alter the contents of the queue. Should the queue be altered, it would be impossible to perform error recovery in case the checkpoint operation is to be rolled back due to an error in a posterior stage. Moreover, if the intent is to simply take a snapshot of the system, a destructive method is entirely inadequate as it will adversely affect the application’s execution after the snapshot is taken.
One method to obtain the contents of the receive queue is to use the `read` system call in a similar way as applications, leveraging the native kernel interface to read directly from a socket. To avoid altering the contents of the queue by draining the data off, this can be done in “peek” mode, which only examines the data but does not drain the queue. Unfortunately, this technique is incomplete and will fail to capture all of the data in the network queues with TCP, including crucial out-of-band, urgent, and backlog queue data.

Another approach is to examine the socket directly and read the relevant data by traversing the socket buffers at a low level. However, the receive queue is of asynchronous nature and is tightly integrated with the implementation of the TCP/IP protocol stack. Reading the chain of buffers requires deep understanding of the relevant kernel mechanisms as well as interpretation of protocol specific information. The result is a prohibitively intricate and non-portable approach.

To get around this we adopt the approach described below, that handles the restore of a socket’s receive queue. In particular, we read the data off the socket using the standard `read` system call, while at the same time injecting it back into the socket. The data ends up attached to the socket as if it has just been restored. Effectively this means that even though the receive queue was modified, the application is still guaranteed to read this data prior to any new data arriving on the network, similar to other restored data.

The kernel does not provide interfaces to insert data into the receive queue, and doing so requires intimate knowledge of the underlying network protocol. This difficulty is overcome by observing that it is sufficient that the application consumes the restart data before any newer data that arrives to the socket. We therefore allocate an alternate receive queue in which this data is deposited. We then interpose on the socket interface calls to ensure that future application requests will be satisfied.
with this data first, before access is made to the main receive queue. Clearly, the checkpoint procedure must save the state of the alternate queue, if applicable (e.g., if a second checkpoint is taken before the application reads its pending data).

Technically, interposition is realized by altering the socket’s dispatch vector. The dispatch vector determines which kernel function is called for each application interface invocation (e.g., `open`, `write`, `read` and so on). Specifically we interpose on the three methods that may involve the data in the receive queue: `recvmsg`, `poll` and `release`. Interposition only persists as long as the alternate queue contains data; when the data becomes depleted, the original methods are reinstalled to avoid incurring overhead for regular socket operation.

Interposing on `recvmsg` is required in order to use the alternate queue as the source for the data, rather than the original queue. The `poll` method is included since it provides asynchronous notification and probing functionality by examination of the receive queue. Finally, the `release` method is important to properly handle cleanup (in case the data has not been entirely consumed before the process terminates).

Extracting the contents of the send queue is more involved than the receive queue, as there does not exist a standard interface from which we can leverage, that provides access to that data. Instead the data is accessed by inspecting the socket’s send queue using standard in-kernel interface to the socket layer (which is normally used by protocol code and device drivers). This is accomplished without altering the state of the send queue itself. While the receive queue is tightly coupled to the protocol specifics and roughly reflects the random manner in which the packets arrived, the send queue is more well organized according to the sequence of data send operations issued by the application. For this reason, unlike with the receive queue, reading the contents of the send queue directly from the socket buffers remains a relatively simple and portable operation.
Finally, restoring the state of the send queue is almost trivial: given the re-established connection, the data is re-sent by means of the standard `write` system call. The underlying network layer will take care of delivering the data safely to the peer socket. In the case of migration, a clever optimization is to redirect the contents of the send queue to the receiving pod and merge it with (or append to) the peer’s stream of checkpoint data. Later during restart, the data will be concatenated to the alternate receive queue (of course, only after the latter has been restored). This will eliminate the need to transmit the data twice over the network: once when migrating the original pod, and then again when the send queue is processed after the pod resumes execution. Instead it will merge both into a single transfer, from the source pod to the destination pod.

We now discuss how the protocol specific state is saved and restored. The portion of this state that records the protocol properties is exported to the socket layer and can be accessed by the applications. TCP options that activate and deactivate keep-alive timers (`TCP_KEEPALIVE`), and semantics of urgent data interpretation (`TCP_STDURG`) are two such examples. The saved state includes the entire set of these options, and they are handled in a similar way to the socket options, as discussed before. ([160] contains a representative list of such options).

The remainder of the protocol specific state is internal, and holds dynamic operational data. Unlike accessing the socket layer which is a common practice in kernel modules, access to the protocol’s state requires intimate knowledge of its internals. Restoring such a state entails carefully handcrafted imitation of the protocol’s behavior. If a portable solution is sought, it is notably desirable to identify the minimal state that must be extracted and restored. As discussed before, the minimal state for unreliable protocols is nil, inherently to their nature. We now discuss reliable protocols.
With reliable protocols, the internal state typically keeps track of the dynamics of the connection to guarantee delivery of messages. Data elements are tagged with sequence numbers, and the protocol records the sequence number of data that has been transmitted but not yet acknowledged. Timers are deployed to trigger resubmission of unacknowledged data (on the presumption that it had been lost), and to detect broken connections. Each peer in a connection tracks three sequence numbers: last data sent (sent), last data received (recv) and last data acknowledged by the other peer (acked).

![Diagram](image)

**Figure 8.2** – Non-overlapping and overlapping data queues

An important property of reliable protocols is the following invariant: recv\textsubscript{1} ≥ acked\textsubscript{2} (where the subindices 1 and 2 designate the peers of the connection). The reason is that upon receiving data, a peer updates its recv value, and sends an acknowledgment. Unless the acknowledgment is lost, it will arrive with some small
delay, and then the other peer will update its **acked** value. It follows that a send queue always holds data between **acked** and **sent**—that is the unacknowledged data. The receive queue holds data from some point back in time until **recv**. If \( \text{recv}_1 > \text{acked}_2 \) there will be some overlap between the two queues. This setting is depicted in Figure 8.2. The overlap must be fixed during the restart operation, before the application consumes duplicate data. This can be done by discarding extraneous data from either queue. It is more advantageous to discard that of the send queue to avoid transferring it over the network.

We claim that a necessary and sufficient condition to ensure correct restart of a connection, is to capture **recv** and **acked** values on both peers. This data, along with additional protocol specific information, is located in a protocol-control-block (PCB) data structure associated with every TCP socket. While the PCB concept is ubiquitous to the TCP stack, the details of its layout differ between distinct implementations. It follows that the need to access these fields does not impair the portability of our scheme, but merely requires a trivial adjustment per implementation.

We now show that the minimal state that we need to extract sums up to the aforementioned sequence numbers. Given the discussion above, these values are necessary in order to calculate the extent of the redundant data to be discarded. They are sufficient since the remainder of the data is in the socket queues, and is already handled as described above. It follows that our approach results in a network state checkpoint/restart solution that is almost entirely independent of transport layer protocol. It is optimal in the sense that it requires no state from unreliable protocols, and the minimal state from transport protocols - that portion of the state that reflects the overlap between a send queue and the corresponding receive queue.

Some applications employ timeout mechanism on top of the native protocol, as a common technique to detect soft faults and deadlocks, or to expire idle connections.
It is also used to implement reliable protocols on top of unreliable ones (e.g., over UDP). The application typically maintains a timestamp for each connection, updating its value whenever there is activity involving the connection. Timestamps are inspected periodically, and the appropriate action is triggered if the value is older than a predefined threshold.

It follows that if there is sufficient delay between the checkpoint and the restart, certain applications may experience undesired effect if the timer value exceeds the threshold and expires. We resolve this by virtualizing those system calls that report time. During restart we compute the delta between the current time and the current time as recorded during checkpoint. Responses to subsequent inquiries of the time are then biased by that delay. Standard operating system timers owned by the application are also virtualized. At restart, their expiry time is set in a similar manner by calculating the delta between the original clock and the current one. We note that this sort of virtualization is optional, and can be turned on or off per application as necessary (so that application that strictly require knowledge of the absolute time can operate normally).

ZapC can transparently checkpoint and restart the network state of TCP, UDP and IP protocols, and therefore any distributed application that leverages these widely used protocols. However, some high performance clusters employ MPI implementations based on specialized high-speed networks where it is typical for the MPI libraries to bypass the operating system kernel and directly access the actual device using a dedicated communication library. Myrinet combined with the GM library [115] is one such example. The ZapC approach can be extended to work in such environments if two key requirements are met. First, the library must be decoupled from the device driver instance, by virtualizing the relevant interface (e.g., interposing on the ioctl system call and device-dependent memory mapping). Second, there must be some
method to extract the state kept by the device driver, as well as reinstate this state on another such device driver.

8.4 Evaluation

We have extended DejaView’s checkpoint-restart component to implement a ZapC prototype as a Linux kernel module and associated user-level tools. Our prototype runs on multiple Linux operating system versions, including the Linux 2.4 and 2.6 series kernels. We present some experimental results on various size clusters with both uniprocessor and multiprocessor nodes running real applications to quantify ZapC’s virtualization overhead, checkpoint and restart latencies, and the resulting checkpoint image sizes.

To measure ZapC performance, we used four distributed applications that use MPI (version MPICH-2 [71]) and PVM (version 3.4), representing a range of different communication and computational requirements typical of scientific applications. One of the applications used PVM while the rest used MPI. Each pod is seen as an individual node so each pod runs one of the respective daemons (mpd or pvmd). The configuration details needed to do this for more than one pod on a physical machine, is to specify an ssh port default for each pod in .ssh/config that the daemon will use to initiate contact with other pods. The applications tested were:

1. **CPI** — parallel calculation of Pi provided with the MPICH-2 library that uses basic MPI primitives and is mostly computationally bound.

2. **BT/NAS** [11] — the Block-Tridiagonal systems (BT) from the NAS parallel benchmark that involves substantial network communication along the computation.
3. *PETSc* [12] — a scalable package of PDE solvers that is commonly used by large-scale applications, in particular the Bratu (SFI - solid fuel ignition) example, which uses distributed arrays to partition the problem grid with a moderate level of communication.

4. *POV-Ray* [137] (PVM version) — a CPU-intensive ray-tracing application that fully exploits cluster parallelism to render three-dimensional graphics.

The measurements were conducted on an IBM HS20 eServer BladeCenter, a modest-sized cluster with ten blades available. Each blade had dual 3.06 GHz Intel Xeon\textsuperscript{TМ} CPUs, 2.5 GB RAM, a 40 GB local disk, and Q-Logic Fibre Channel 2312 host bus adapters. The blades were interconnected with a Gigabit Ethernet switch and connected through Fibre Channel to an IBM FastT500 SAN controller with an Exp500 storage unit with ten 70 GB IBM Fibre hard drives. Each blade used the GFS cluster file system [155] to access the shared SAN.

We measured the applications running across a range of cluster configurations. We ran the ZapC Manager on one of the blades and used the remaining nine blades as cluster nodes for running the applications. We configured each cluster node as a uniprocessor node and ran each application except BT/NAS on 1, 2, 4, and 8 nodes. We ran BT/NAS on 1, 4, 9 and 16 nodes because it required a square number of nodes to execute. We also configured each cluster node as a dual-processor node and ran each of the applications on eight of the nodes. Since each processor was effectively treated as a separate node, we refer to this as the sixteen node configuration. Results are presented with each blade running Linux 2.6, specifically Debian Linux with a 2.6.8.1 kernel. Linux 2.4 results were similar and are omitted for brevity.
8.4.1 Virtualization Measurements

We measured ZapC virtualization overhead by comparing the completion times of the applications running on two configurations, which we refer to as Base and ZapC. Base was running each application using vanilla Linux without ZapC, thereby measuring baseline system performance. ZapC was running each application inside ZapC pods. Each execution was repeated five times and the results were averaged over these runs.

Figure 8.3 shows the average completion times of the different benchmarks for different numbers of nodes. The results show that the completion times using ZapC are almost indistinguishable from those using vanilla Linux. Results for larger cluster systems were not available due to the limited hardware that was available at the time of the experiments. However, the results on a modest cluster size show that ZapC does not impact the performance scalability of the applications as the relative speedup of the applications running on larger clusters is essentially the same for both vanilla Linux and ZapC.

The virtualization overhead was in all cases much smaller than even the variation in completion times for each configuration across different runs. This further validates that DejaView’s thin virtualization layer imposes negligible runtime overhead on real applications. The standard deviation in the completion times for each configuration was generally small, but increased with the number of nodes up to roughly 5%. The variations were largely due to differences in how much work the applications allocated to each node from one execution to another.

8.4.2 Checkpoint-Restart Measurements

We measured ZapC checkpoint-restart performance by running each of the four distributed applications and taking ten checkpoints evenly distributed during each appli-
Figure 8.3 – Application completion times on vanilla Linux and ZapC
cation execution. Each checkpoint was a full checkpoint (i.e., non-incremental) with deferred write. We omitted file system snapshots, pre-quiesce, and pre-snapshots, to focus on the performance of saving and restoring only the execution state. We measured checkpoint and restart times for each application as well as the checkpoint image sizes for each application. Due to the limited number of nodes available, restarts were done using the same set of blades on which the checkpoints were performed.

Figure 8.4 shows the average checkpoint times across all ten checkpoints for each application. This is the time from the invocation of the Manager by the user until all pods have reported “done”, as measured by the Manager. The time includes the time to write the checkpoint image of each pod to memory and represents the time that the application needs to be stopped for the checkpoint to occur, which provides an indication of how frequently checkpoints can be done with minimal impact on application completion time. This does not include the time to flush the checkpoint image to disk, which can be done after the application resumes execution and is largely dependent on the bandwidth available to secondary storage.
Figure 8.4 shows that checkpoint times are all sub-second, ranging between 100 ms and 300 ms across all four applications. For a given application and a given cluster size, the standard deviation in the checkpoint times ranged from 10% to 60% of the average checkpoint time. For all applications, the maximum checkpoint time was no more than 200 ms of the respective average checkpoint time. For all checkpoints, the time due to checkpointing the network state as seen by the respective Agent was less than 10 ms, which was only 3% to 10% of the average checkpoint time for any application. The small network-state checkpoint time supports the motivation for saving the network state as the first step of the distributed checkpoint, as discussed in §8.2.

Figure 8.5 shows the restart times for each application as measured based on the time it took to restart the respective application from its checkpoint image. This is the time from the invocation of the Manager by the user until all pods have reported “done”, as measured by the Manager. The time assumes that the checkpoint image has already been preloaded into memory and does not include the time to read the

<table>
<thead>
<tr>
<th>No. of Nodes</th>
<th>CPI</th>
<th>PETSc</th>
<th>POV-Ray</th>
<th>BT/NAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Figure 8.5 – Distributed application restart times
image directly from secondary storage. To provide a conservative measure of restart time, we restarted from a checkpoint image taken in the middle of the respective application’s execution during which the most extensive application processing is taking place.

Figure 8.5 shows that restart times are all sub-second, ranging between 200 ms and 700 ms across all four applications. The restart times are longer than the checkpoint times, particularly POV-Ray which takes the longest time to restart. Restart times are longer than checkpoint times in part because additional work is required to reconstruct the network connections of the participating pods. The network-state restart time in most cases ranges between 10 ms and 200 ms.

Figure 8.6 shows the size of the checkpoint data for each application running with different numbers of cluster nodes. For a given application and cluster configuration, the checkpoint image size shown is the average over all the ten checkpoints of the largest image size among all participating pods. Since the checkpoints of each pod largely proceed in parallel, the checkpoint size of the largest pod provides a more
useful measure than the total checkpoint size across all pods. In most cases, the checkpoint size of the largest pod decreases as the number of nodes increases since the workload assigned to each node also decreases. The checkpoint size for CPI goes from 16 MB on 1 node to 7 MB on 16 nodes, the checkpoint size for PETSc goes from 145 MB on 1 node to 24 MB on 16 nodes, and the checkpoint size for BT/NAS goes from 340 MB to 35 MB on 16 nodes, an order of magnitude decrease. Only POV-Ray has a relatively constant checkpoint size of roughly 10 MB. These results suggest that the maximum pod checkpoint image size scales down effectively as the size of the cluster increases. This provides good performance scalability for larger clusters since checkpoint size can be an important factor in the time it takes to read and write the checkpoint image to disk.

For the applications we measured, the checkpoint size of the largest pod was much larger than the portion of the checkpoint size due to network-state data. The size of the network-state data was only a few kilobytes for all of the applications. For instance in the case of CPI, the network-state data saved as part of the checkpoint ranged from 216 bytes to 2 KB. Most parallel applications are designed to spend significantly more time computing than communicating given that communication costs are usually much higher than computation costs. Therefore, at any particular point in time, it is most likely that an application has no pending data in its network queues as it is likely to have already been delivered. It follows that the application data largely dominates the total checkpoint data size. Our results show that the size of application data in a checkpoint image can be many orders of magnitude larger than the size of network data.
8.5 Summary

In this chapter, we presented ZapC, a system for transparent coordinated checkpoint-restart for distributed applications on commodity clusters. ZapC provides three key mechanisms. First, it leverages DejaView’s checkpoint-restart mechanism to migrate applications across different clusters while utilizing available commodity operating system services. Second, it introduces a coordinated, parallel checkpoint-restart mechanism that minimizes synchronization requirements among different cluster nodes to efficiently perform checkpoint and restart operations. Third, it integrates a network state checkpoint-restart mechanism that leverages standard operating system interfaces as much as possible to uniquely support complete checkpoint-restart of network state in a transport protocol independent manner, for both reliable and unreliable protocols. Our experimental results on a range of distributed scientific applications demonstrate that ZapC incurs negligible overhead and does not impact application scalability. Furthermore, they show that ZapC can provide fast, sub-second checkpoint and restart times of distributed applications. The results also show that network-state checkpoint and restart accounts for a very small part of the overall time, suggesting that our approach can scale to many nodes.
Chapter 9

Desktop Power Management

With the growing computer-infested IT infrastructures worldwide, conserving energy has become increasingly important for corporations from both the environmental and economical perspectives. Although modern hardware offers power-saving sleep states, in practice many desktops and servers are always kept powered on even during periods of idle time [6]. Suspending a desktop disrupts ongoing connections and makes it inaccessible over the network, making users and administrators reluctant to do so because their applications must maintain network connections [5], or because many idle periods are short and scattered [41]. To overcome this barrier, computers need a way to maintain network connectivity while in standby state. In this chapter, we leverage DejaView’s checkpoint-restart mechanism to enable mobility of individual applications, allowing significant energy saving in desktops by migrating networked applications out of sleeping hosts to ensure continuous network presence for them.

Several approaches have been proposed to address this issue, including network proxies, specialized hardware, and rewritten software, but they all have limitations that prevent them from being widely deployed. A network proxy operates on behalf of a sleeping computer by filtering and responding to network protocols, and emu-
lating applications behavior. However, the proxy functionality comes at the cost of greater complexity as it requires developing often complex application-specific stubs. Specialized hardware is not readily available and deployable across and enterprise, and rewriting all network applications is prohibitively expensive, rendering both approaches also impractical.

Building on DejaView’s virtualization and checkpoint-restart tools, we present NetCont, a system for allowing networked applications to maintain continuous network presence even when the computer on which they run remains in standby state. In NetCont, a dedicated server is used to host network state and network applications, allowing idle or unattended computers to enter low-power suspended state while keeping their connections active and their networked applications available. When a user’s computer prepares to suspend, all network state and connections are migrated to a server where they continue to run and provide network presence, until they are migrated back when the user’s computer resumes from suspend.

NetCont leverages application virtualization to encapsulate a user’s applications within containers, allowing to migrate individual applications by taking a checkpoint on one computer and restarting on a different computer. Unlike the approaches that employ specialized proxy servers with hand-tailored logic to match specific applications, NetCont operates transparently and relies on the applications themselves to maintain their network presence by migrating them and allowing them to execute without interruption. Unlike approaches that use virtual machines, NetCont migrates only the state of networked applications, not the entire operating system instance or the entire desktop.

We have implemented a NetCont prototype in Linux and demonstrate its effectiveness in transparently providing network presence for a range of unmodified real applications. Our results show that NetCont imposes negligible runtime over-
head and provides fast migration times without degrading the user’s experience, and without any changes to the network infrastructure, computer hardware, applications, libraries or operating system kernel. NetCont’s lightweight operation scales well, and its fast migration times allow it to exploit even short idle times.

9.1 Architecture Overview

NetCont allows idle computers to enter low power suspend state while keeping their network-facing applications active and their network services available. To accomplish this, NetCont introduces a dedicated Network Connection Server (NCS) that maintains network presence for sleeping hosts, enabling them to be automatically resumed when their services are required. In addition to maintaining a passive network presence, NetCont combines application virtualization and mobility to quickly migrate active networked applications temporarily to the NCS, where they continue to run while their origin host is asleep. When the host resumes normal operation, the application quickly migrate back to it. The operation of NetCont is summarized in Figure 9.1.

Figure 9.1 – NetCont operation: (a) All fours desktops are in use, running a mix of regular and networked applications (in ellipse and circle containers, respectively), and the NCS is idle. (b) Three desktops are in standby state, and the NCS owns their respective IPs, network state and networked applications.
A key property of NetCont’s ability to support active networked applications is that it operates at the granularity of these individual applications, rather than include the entire desktop and operating system instance. Moreover, migration targets only networked applications that have active connections. The execution fingerprint of individual applications in terms of processor, memory and network usage is significantly lower than an entire machine. This allows NetCont to pack many application instances from multiple sleeping hosts into a single NCS instance. By moving the entire running application, we avoid the need for per-application logic within the NCS, as the unmodified application will continue to execute normally after migration. By skipping non-networked applications, as well as the underlying operating system, we enhance the NCS capacity because we reduce both the amount of state stored on it as well as its load. The combined effect is increased energy savings.

This architecture has four additional useful properties. First, an NCS is easy to install: it runs on commodity hardware and uses a standard operating system. Second, it is efficient and lightweight: a mid-size NCS can typically cater to applications from hundreds of sleeping hosts. Third, it is scalable and can be deployed incrementally by adding more NCS instances as needed. Fourth, it is transparent, and does not require any changes to the user’s desktop, its operating system, the applications, or the network infrastructure. NetCont also does not change the user’s experience, and is completely transparent to remote network peers.

When a user’s computer prepares to transition to low power state, NetCont performs a number of steps. First, it arranges to transfer its network state to the NCS. The network state includes generic state such as the IP address, ARP table entries, DHCP lease state, and associated SSID for wireless networks. Second, it registers the set of ports the NCS should monitor for traffic that should cause the NCS to wake the computer from its low power state. Common examples include
ports that are used for remote access of the machine (SSH, RDP, VNC) or are used to perform remote management (Anti-Virus, System Scan). Third, it migrates all the active networked applications with their established network connections to the NCS, where they continue with their execution seamlessly. Finally, when network migration completes, the user’s computer concludes its transition to sleep state.

The NCS impersonates the host to transparently maintain network reachability to the host while the host is asleep. Because the NCS owns the network IP, it leverages the standard network stack to automatically respond to generic traffic such as ARP requests, DHCP lease renewals, and ICMP packets. Because the NCS owns the active network connections, the standard network stack also handles transport layer traffic such as TCP keep-alive and packets retransmit. Because the NCS keeps the networked applications which run unmodified, application layer traffic is handled natively by the applications themselves.

The NCS maintains the sleeping computer’s network presence in two ways. First, it responds to incoming network traffic as necessary, whether automatically through the network stack, or by letting the corresponding application receive and process the data. For example, remote users will still be able to ping the machine. Second, if incoming traffic is detected on one of the registered ports, it wakes up the sleeping computer through the use of the Wake-on-LAN standard. The NCS ignores all other network packets destined for the sleeping computer by silently dropping them.

By performing these functions, NetCont ensures that the transition of the computer into sleep state is transparent to remote hosts on the network. The NCS carries out these duties until the host wakes up, e.g., because the user returns to work on it, or because an incoming packet requires its attention. When the host wakes up, NetCont migrates the network state and the active networked applications back to the user’s computer.
In addition to waking the machine when traffic is detected on registered ports, NetCont will wakeup the sleeping host machine when networked programs running on the NCS require the use of programs that have remained on the sleeping server, as will be discussed in §9.2. Additionally, if both the host and NCS are Internet facing, many attacks, such as brute force SSH password attacks are common and can prevent the computer from sleeping. In these situations, NetCont can migrate the entire SSH service to the NCS, instead of just registering the SSH port, and only wakeup the host computer when the SSH login completes successfully.

9.2 Application Containers

At the heart of NetCont’s operation is the ability to migrate only the networked applications. However, migration of individual applications belonging to a coherent desktop framework raises two key challenges. First, applications are generally inter-related through process dependencies such as parent, child, and sibling relations, inter-process communication mechanisms such as pipes, and shared resources such as inherited file descriptors. These make it impossible to transparently migrate an individual application. Second, desktop applications are integrated with the desktop through common services such as the task-bar, clipboard, and the system logger. Furthermore, applications can launch other applications in response to user actions, e.g., a mouse click on a hyper-link in an email client will open the responding page in a web browser.

NetCont leverages previous work on Apiary [132, 136] to decompose a user’s desktop into multiple application containers, one for applications that will never be migrated and individual containers for each independently migratable application. Each application container is an independent appliance that provides all the system
services that the applications need to execute. Using containers to run the applications decouples the applications from the underlying operating systems instance, enabling them to be migrated to and from the desktop computer and the NCS. By isolating each migratable application into its own container, NetCont provides a fine grained approach to the migration of the networked applications.

To provide traditional desktop semantics, NetCont integrates these containers at the display and file system levels, while also enabling applications in separate containers to interact with each other through normal inter-process communication mechanisms. NetCont also maintains a traditional desktop experience. Users launch applications from a menu or from within each other, switch among available applications using a task-bar, interact with their applications using standard input devices, and have a single display with integrated window system and clipboard for all the applications. For example, an IM client in a migrating container and a word processor in the non-migrating container look and feel like part of the same consistent desktop, with all normal windowing functions and cut-and-paste operations working seamlessly across all applications.

To accomplish this, NetCont leverages DejaView’s virtualization architecture to provide a virtual execution environment, a virtual display system that provides a virtual display server and viewer, and a shared network file system that provides a common file system for the containers and can be shared between the desktop host and the NCS. Additionally, NetCont runs one manager daemon on the host, outside of the containers, and one connector daemon within each container to help integrate the containers together correctly.

The use of a virtual display is essential for enabling the migration of only a subset of the desktop’s applications. Without it, all applications share a single display, and migration of any single container without the others becomes impossible. The virtual
display decouples the display state of each application from the underlying hardware and redirects the display output to the NetCont viewer. The display server in each container maintains all persistent display state. The viewer is simple and stateless.

To the user, NetCont presents a coherent desktop experience with the normal usage metaphors in four dimensions, namely the display, the application launch menu, the task-bar, and inter-application interaction. For the display, NetCont integrates the views from each container into a single coherent view. More specifically, each display provides an alpha channel color for its desktop background and the viewer stacks them in a series of layers where objects higher in the stack obscure elements of objects below. The views are ordered based on the currently used applications. This order changes as the user switches between applications. All the virtual displays use the same resolution and they completely overlap such that windows may appear anywhere on the composited desktop display.

For the launch menu, NetCont integrates the list of all the applications that users are able to launch from each container. The viewer collects this information by querying the connector process in each container. The connector process constructs this list within its container but does not draw the menu on screen. Instead, it delegates the data to the viewer, which integrates this information into its own menu. When a user selects a program from the viewer’s menu, the viewer instructs the connector to execute the program within its container.

To implement a coherent task-bar to switch between applications, the connector also constructs an enumeration of all the application running inside the container and reports this information to the viewer. The viewer integrates this information into a single task-bar that provides buttons that correspond to applications windows. When the task-bar is used to change which window is focused, the viewer instructs the corresponding connector to bring the respective window to the foreground.
Running the applications within separate containers can prevent the applications from integrating together in a normal manner. By default, programs within separate containers can interact with each other only through standard network communication mechanisms. However, this could prevent users from using their desktop applications effectively. To solve this problem, NetCont integrates the containers at both the application execution and inter-process communication levels.

NetCont enables an application in one container to launch another application in another container. For example, a user can click on a link appearing in her IM client, located in one migratable container, and have it launch the web browser located in another migratable container. This is implemented by including wrapper programs in each container for programs provided by other containers that are made available for execution. For example, a wrapper for /usr/bin/firefox in each container can delegate the options passed to it to the manager process, when invoked. The manager is then able to execute the requested command in the appropriate container for that program, causing it to execute in a transparent manner to the user.

The wrapper mechanism also enables networked applications that have been migrated to the NCS to still make use of services provided by applications remaining on the sleeping host computer. For instance, a networked program that is in the process of downloading a file will want to virus check it when the download completes. When a networked application that has been migrated to the NCS executes the wrapper program, NetCont wakes the host computer, migrates the networked application containers temporarily back to the host so that the program can communicate with the local service. When the program terminates, NetCont migrates the networked application containers back to the NCS and allows the host to suspend itself.

In terms of inter-process communications, NetCont hides the segregation of applications into isolated containers by using the manager process as a proxy to such
services like SysV-IPC and D-Bus. This enables global desktop state to be shared between containers. For example, to handle the clipboard, each container’s connector monitors the local clipboard and delegates updates to that container’s clipboard to the manager process. The manager process propagates updates to all the other containers. Similarly, when an application writes to the system logger, the connector intercepts the message inside the container, and hands it over to the manager process. The manager process passes it to the main container that hosts the system logger.

9.3 Application Migration

NetCont migrates networked applications by capturing the state of an entire networked container on one host and restoring that state on another host. NetCont’s migration can be thought of in terms of two logical components: a stand-alone checkpoint-restart mechanism that can save and restore the non-network application state, and a network checkpoint-restart mechanism that can save and restore the network state, including the existing network connections.

The network state consists of two components: host network state and network connections state. Host network state includes link layer and network layer states, such as the host IP address, ARP table entries, DHCP lease state, and routing information. This state is global to the entire host and independent of the applications. Network connections state include transport layer state, i.e., the state related to established network connections between applications and their peers. The state of the session, presentation, and application layers need not be explicitly saved and restored since it is part of the application logic which is included in the stand-alone state.

Because the network state includes the computer’s IP address, the operation of NetCont is constrained to occur only within a single routing domain of the network,
i.e., a single IP subnet. For simplicity, we assume a shared storage infrastructure between the computers and the NCS. In this case, file system state is not generally saved and restored as part of the container checkpoint image to reduce checkpoint image size.

There are two key challenges when migrating networked applications with established connections between different hosts. First, it is necessary to replicate the network and protocol stack states on the destination host. In particular, the state of all established connections must be explicitly recorded during checkpoint and restored during restart. This is quite different than migration of virtual machines in which the entire operating system instance is moved as a whole, implicitly including the network state. Saving the network state explicitly is hard because this state is tightly coupled to the underlying operating system and the specifics of the kernel’s networking subsystem.

A naive approach to restore the network state is to manually craft the internal kernel data structures, populate them with suitable data, and carefully adjust the protocol stack state. However, this approach is complex and requires intimate knowledge of, integration with, and adjustments to the networking subsystem and protocol stack implementation [82, 121]. Instead, NetCont takes a simpler approach that decouples the reconstruction of the connection from the details of the underlying implementation by reusing as much as possible existing standard operating system interfaces.

To understand how this works, we first observe that the state of every network connection can be divided into two pieces: a base connection state that includes the permanent attributes of the connection, such as the protocol type, source, and destination addresses, as well as the send and receive buffers sizes; and the dynamic connection state that reflects the transient attributes of the connection state, includ-
ing the data stored in the send and receive buffers and protocol specific state such as TCP sequence numbers, timestamps, and transmit window size, to name a few.

Building on this observation, NetCont divides the network restore work into two steps. For each saved connection, NetCont first restores the base state by creating a new connection with the desired static attributes. This is done in user-space using standard operating system interfaces. Next, for each connection, NetCont restores the dynamic state by reinstating the respective transient attributes of the connection. This is performed in the kernel, since it may require direct changes to some in-kernel protocol specific state.

A second challenge when migrating networked applications with established connections is that any communicating peers connected to the applications do not cooperate with the migration and must remain unaware of it. Thus, the migration must occur in a manner that is both atomic and transparent to the peers. Atomicity ensures that the peers observe a consistent network state on a single host, rather than split state between the origin and the destination host. Transparency ensures that peers may continue to run normally without requiring any special action post-migration.

To satisfy the requirement for atomicity, NetCont blocks the network traffic to and from the origin and destination hosts for the duration of the migration (but permits direct communication between them). Blocking outgoing packets prevents state changes or partial state from becoming visible to the peers prematurely. Dropping incoming packets prevents the migrating network state from changing while it is being saved or restored.

To satisfy the requirement for transparency, NetCont uses the ARP protocol used to map link layer hardware addresses to IP addresses. Similar to other live migration approaches [35, 82, 147], NetCont generates an unsolicited ARP reply from the destination host once the migration completes successfully, to update the
ARP cache of the peers (or the nearest intermediate switch or router), informing them of the new mapping.

9.3.1 Checkpoint and Restart Overview

We now describe the algorithms for checkpoint and restart used by NetCont to migrate networked applications between the user’s desktop and the NCS. Both algorithms handle the network state before the container (and applications) state. Doing so enables more concurrent operation by overlapping the dominant component of the operation, namely the stand-alone restart time, with the time it may take the network layer to recover from the temporary blockage. Re-enabling network traffic at the earliest, as soon as the network state is restored, enables the network to retransmit pending data in the queues and acknowledge incoming packets concurrently with the stand-alone restart. This not only reduces the possibility of undesired protocol timeouts, but also enables the protocol’s flow control mechanism to get up to speed.

Checkpoint: To checkpoint the networked applications on the origin hosts, NetCont executes Algorithm 9.1. The algorithm suspends the container to prevent the applications’ states from being altered during the checkpoint. It also suspends the network activity to and from the host to prevent the network state from changing, by leveraging standard network packet filtering frameworks such as netfilter [118], to block the host’s network address(es) entirely. Once the container and the network are suspended, NetCont saves the host’s network state. It first saves the base connection state of each established connection on the host, followed by the dynamic connection state of each connection. After saving the network state, NetCont performs the stand-alone checkpoint, and thereafter the container is shutdown and destroyed.
Chapter 9. Desktop Power Management

Algorithm 9.1: Checkpoint

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BlockNetwork(IP);</td>
</tr>
<tr>
<td>2</td>
<td>SuspendContainer()</td>
</tr>
<tr>
<td>3</td>
<td><strong>foreach</strong> connection <strong>conn</strong> <strong>do</strong></td>
</tr>
<tr>
<td>4</td>
<td>SaveConnectionBase(conn)</td>
</tr>
<tr>
<td>5</td>
<td>SaveConnectionDynamic(conn)</td>
</tr>
<tr>
<td>6</td>
<td><strong>end</strong></td>
</tr>
<tr>
<td>7</td>
<td>SaveContainer(cont)</td>
</tr>
<tr>
<td>8</td>
<td>DeleteContainer(cont)</td>
</tr>
</tbody>
</table>

Algorithm 9.2: Restart

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BlockNetwork(IP);</td>
</tr>
<tr>
<td>2</td>
<td><strong>foreach</strong> connection <strong>conn</strong> <strong>do</strong></td>
</tr>
<tr>
<td>3</td>
<td>RestoreConnectionBase(conn)</td>
</tr>
<tr>
<td>4</td>
<td>RestoreConnectionDynamic(conn)</td>
</tr>
<tr>
<td>5</td>
<td><strong>end</strong></td>
</tr>
<tr>
<td>6</td>
<td>UnBlockNetwork(IP);</td>
</tr>
<tr>
<td>7</td>
<td>CreateContainer(cont)</td>
</tr>
<tr>
<td>8</td>
<td>RestoreContainer(cont)</td>
</tr>
</tbody>
</table>

**Restart:** To restart the applications on the destination host, NetCont executes Algorithm 9.2, which follows a flow similar to the checkpoint algorithm. The algorithm blocks the host’s network to protect against packets they may arrive prior to completely restoring the corresponding connection. If such packets arrive before the connection exists, they will be rejected, leading the peer to drop the connection; if the connection exists but not fully setup yet, they may trigger an error and fail the connection locally. The network remains disabled until the entire network state is successfully restored. NetCont now restores the host’s network state, first the base state followed by the dynamic state of all connections, and re-enables the network. It then creates a new container for the migrated applications, and performs the standalone restart to restore the applications into this container. The final notch of the procedure is to unblock the host’s network and generate an unsolicited ARP packet,
to advertise that the network address has moved to a new location and is now bound
to a distinct hardware address.

While blocking the network during a migration means that incoming packets are
silently dropped, it is a safe practice because reliable transport protocols such as
TCP are designed to retransmit lost packets. However, it may adversely impact
performance since it changes the peer’s view of the connection. In TCP, timeouts
due to unacknowledged data progressively increase the retransmission timeout and
reduce the transmission window size. After a migration, a connection is likely to stall
for a considerable period as the peer awaits the next timeout expiry before it resends
data. Sending new (or duplicate) data to the peer can alleviate the problem, but will
still suffer TCP slow-start behavior before data transfer rate climbs back up. In this
case, not even TCP’s fast-retransmit mechanism can help because it does not apply
to timed-out packets.

To avoid this behavior, NetCONT cleverly exploits TCP’s existing flow control
mechanisms to inform the peers, prior to migration, that they should not send any
more data. To do so, NetCONT forces each connection to advertise a window of
size zero to the respective peers, ensuring that the peers will not attempt to send
more data, and, in turn, will not face timeouts due to unacknowledged data. Peers
will continue to send probes and may eventually abort the connection, but typical
timeouts are orders of magnitude longer than actual migration time. (This also
holds for application level timeouts, which are, too, much larger than the migration
time). When the migration completes, NetCONT re-advertises the original saved
window from before the migration, so that the connection can resume under the
same condition as it was left off.


9.3.2 Base Connection State

In describing how NetCont migrates network connections states, it is convenient to refer to sockets, which are the primary abstraction of communication endpoints from the applications’ point of view. A socket is associated with a network protocol upon creation. Applications can bind a socket to an address, connect to an address, listen for incoming connections, and accept new connections. Once a socket is connected, the application can exchange data with the peer through it.

**Checkpoint:** To checkpoint the base connection state, NetCont iterates over all the sockets in the container and records their static attributes in an array. Entries in the array are tuples of the form $\langle$source, target, state, type$\rangle$. The array holds one entry for each socket, including for partly-established connections, such as TCP connections that have not completed their three-way handshake with their peers. The source and target fields hold the corresponding source and destination network addresses and port numbers. The state fields holds static attributes from the socket layer, such as the send and receive buffers sizes and the socket options, and from the protocol layer, such as protocol options and static protocol parameters negotiated with the peer. In this way the array records the base connection state in a manner that provides enough information regarding how to correctly reconstruct all the sockets (i.e., network connections) during restart.

The type field indicates the type of the connection. The possible values are listed in Table 9.1. Sockets of listening type are set to accept incoming connections, so they have only their source filled. The pending type indicates a connection that was made from a peer to a listening socket, but that has not yet been claimed by an application (e.g., through the accept system call). Finally, the types outgoing and incoming indicate connections that are still in a transient, incomplete state.
<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>listening</td>
<td>end-point dedicated to accept connections</td>
</tr>
<tr>
<td>established</td>
<td>established connection</td>
</tr>
<tr>
<td>pending</td>
<td>unclaimed established connection</td>
</tr>
<tr>
<td>outgoing</td>
<td>transient outgoing connection</td>
</tr>
<tr>
<td>incoming</td>
<td>transient incoming connection</td>
</tr>
</tbody>
</table>

Table 9.1 – Network connections states

**Restart:** NetCont aims to restore the base connection state of all sockets from user-space. This is accomplished using a novel approach that combines standard socket system calls with a ubiquitous packet filtering framework [118] to reconstruct network connections without the cooperation of the original peers. Instead of relying on the real peer, NetCont introduces a fake peer, co-located on the same host as the socket being restored. It then cleverly exploits packet filtering to redirect the network traffic so that outgoing packets to the real peer will reach the fake peer, and incoming packets from the local fake peer will appear to have arrived from the real peer. In this manner, we are able to create sockets whose state indicates that they are connected to the real peers, however in practice they really communicate with the local fake socket.

To restore the source address of a socket, NetCont uses the standard `bind()` socket system call, which associates a socket with a given local address. However, the network semantics forbid binding multiple sockets to a single local address. Thus, `established` sockets must be restored before `listening` sockets. As we describe below, `listening` sockets are necessary to successfully restore `pending` sockets. Therefore, NetCont orders the sockets such that it first restores `established` sockets, followed by `listening` sockets, and finally `pending` sockets.

The steps to restore the state of `established` sockets are given in Algorithm 9.3. First, NetCont creates a temporary socket to listen for incoming connections. The
Algorithm 9.3: RestoreEstablishedConnection

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>listen ← temporary listening socket</code></td>
</tr>
<tr>
<td>2</td>
<td><code>foreach tuple of established connection do</code></td>
</tr>
<tr>
<td>3</td>
<td><code>sock ← new socket</code></td>
</tr>
<tr>
<td>4</td>
<td><code>AddDNAT(tuple.dst → listen)</code></td>
</tr>
<tr>
<td>5</td>
<td><code>bind(sock, tuple.src)</code></td>
</tr>
<tr>
<td>6</td>
<td><code>connect(sock, tuple.dst)</code></td>
</tr>
<tr>
<td>7</td>
<td><code>dst ← accept(listen)</code></td>
</tr>
<tr>
<td>8</td>
<td><code>DelDNAT(tuple.dst → listen)</code></td>
</tr>
<tr>
<td>9</td>
<td><code>close(dst)</code></td>
</tr>
<tr>
<td>10</td>
<td><code>end</code></td>
</tr>
<tr>
<td>11</td>
<td><code>close(listen)</code></td>
</tr>
</tbody>
</table>

The role of this socket is to accept the connections from the host that, while originally destined for the original peers, arrive back to the host after having been redirected. Next, NetCONT iterates over all the tuples that correspond to established sockets. For each tuple, it first installs a suitable destination network address translation (DNAT) packet filtering rule to divert all network traffic destined for the target address to instead reach the listening socket. The network filter will intercept network packets from source to target and rewrite their destination address. NetCONT then uses standard socket system calls to create a new socket, bind the new socket to the local source address, and connect it to the target address.

Because of the DNAT rule, the connection request will be delivered to the locally listening socket instead of the original peer, and NetCONT can accept this connection, locally, thereby creating a temporary socket. Because the DNAT re-write is transparent to the transport layer, the base connection state of the socket being restored is correctly set, reflecting a connection to the original peer. The DNAT rule and the temporary (accepted) socket are then discarded as they are no longer needed. Once all the tuples have been processed, the listening socket is then discarded, too. The operation of the algorithm is illustrated in Figure 9.2.
Figure 9.2 – Restore an established connection: socket $SRC$ is to be reconnected to the peer $DST$. Network traffic is blocked, and a DNAT rule is in place. As we connect the socket to the peer (a), the request is redirected back to $LISTEN$ socket (b). Once the connection is accepted (c), the DNAT rule is removed and $TMP$ is discarded. Our socket is connected to the peer as desired (d).

To restore the state of pending connections, NetCont needs to create sockets that correspond to incoming connections appearing to have originated from the real peer. We again use packet filtering to fake suitable incoming connections using temporary sockets and traffic redirection. In this case, packets are re-written to appear to arrive from the real peer with source network address translation (SNAT), which modifies the source network the filtered packets. The SNAT re-write, much like the DNAT, is transparent to the transport layer, allowing the resulting socket to be set as desired.

The steps to restore state of pending connections are given in Algorithm 9.4. (The algorithm is executed after all the listening sockets have been restored and
Algorithm 9.4: RestorePendingConnection

1. foreach tuple tuple of pending connection do
2.   AddSNAT(tuple.src → tuple.tmp)
3.   listen ← listeners[tuple.src]
4.   dst ← new socket
5.   connect(dst, listen)
6.   DelSNAT(tuple.src → tuple.tmp)
7.   close(dst)
8. end

stored in are the array Listeners[]. NETCONT iterates over all the tuples that correspond to pending sockets. For each tuple, NETCONT installs a suitable SNAT rule to substitute the source address of packets destined for that listening socket with the real peer address. It then uses standard socket system calls to create a temporary socket and connect it to the listening socket (which was created a-priori) that matches the source address indicated by the tuple.

Because of the SNAT rule, the connection request will arrive to the listening socket appearing to have originated from the real peer. Note that the connection is intentionally not explicitly accepted so that it remains unclaimed by the application. The SNAT rule and the temporary socket are discarded as they are no longer needed. The operation is illustrated in Figure 9.3.

9.3.3 Dynamic Connection State

The dynamic state of a connection, which reflects the transient attributes, consists of both the data stored in the send and receive buffers, as well as protocol stack parameters. Not all protocol stack parameters need to be included in the saved state. Only parameters that are either visible to the peer or that came from the peer, such as sequence numbers and timestamps of the host and the peer, respectively, are important. In contrast, parameters private to the host can be safely ignored.
Figure 9.3 – Restore a pending connection: a pending connection from *SRC* is to be restored. Network traffic is blocked, and an SNAT rule is in place. As we connect a temporary socket (a) to a listening socket, SNAT redirects the request to seem as if it came from *SRC* (b). Once the connection is established (c) but not yet accepted, the SNAT rule is removed and both *TMP* and *LISTEN* are discarded (d).
Examples include parameters that are constantly recalculated such as TCP’s RTT value, TCP timers states, and any internal bookkeeping.

To save and restore dynamic connection state, it is necessary to have access to the respective kernel data structures. Because this state is generally only accessible within the kernel, NetCont relies on a kernel module for such access, similarly to other approaches [82, 97, 113]. NetCont employs techniques similar to theirs for the data stored in the send and receive buffers, by peeking at the buffers to save the data, and populating them to restore the data. However, while this method can also restore the protocol parameters, it does not work for state that is not specific to a particular socket, such as the global kernel time used to generate TCP timestamps, since such state is also used elsewhere and may not be arbitrarily modified.

To address this challenge in a generic way, NetCont takes a different approach that does not overwrite the internal protocol state of sockets; at its core is the observation that it is the information visible to and from the peer that matters, rather than how it is locally stored. For example, the peer will expect (and generate) TCP timestamps post-migration that are consistent with those before migration, regardless of the actual kernel time at the new host. Building on this understanding, we leverage network packet filtering to manipulate packets on the fly to ensure that the protocol state they carry is correct, independently of the underlying dynamic connections state in the kernel.

The mechanism works as follows. Once the base connection state of a socket is restored, NetCont calculates the differences between the saved values and the (arbitrary) new values of the socket’s dynamic state. Then, a packet mangling filter is setup for the socket that intercepts all of the socket’s traffic. The filter adjusts each intercepted packet by adding the computed deltas to the respective values in the protocol header of that packet. Revisiting the previous example, NetCont computes
the delta between the saved kernel time (from before the migration) and the current
kernel time, and the filter will add that delta to the timestamp field of every outgoing
packet. The method also works for other protocol parameters in a similar manner.

In principle, it may also be necessary to mangle incoming packets to adjust state
originating from the peer, such as the sequence numbers managed by the peer. How-
ever, this can be avoided through clever packet mangling of incoming packets during
TCP handshake phase only (with the local, fake peer) to set the initial values of
such parameters as desired. For instance, the sequence number from the peer can be
adjusted in the SYN or SYNACK packets so that the value picked by the restored
socket be correct and not require further adjustments. The end result is a robust
method for restoring dynamic connection state of sockets that it more portable, and
significantly reduces the need for intimate knowledge with the networking subsystem
internals.

9.4 Evaluation

We have implemented a NetCont prototype for Linux desktop environments. Using
this prototype, we present experimental data that quantifies the performance of
NetCont when running a range of desktop applications that use persistent network
connections. We focus on quantifying the power consumption, memory requirements,
and migration time. We already showed in Chapter 5 that the overhead of the virtual
execution environment and is quite small.

The prototype consists of a loadable kernel module and a set of user-space utilities
for off-the-shelf Linux 2.6 kernels that provide the virtual execution environment and
the ability to migrate networked containers between desktops and the NCS. For packet
filtering and mangling, we use Linux’s netfilter [118], that instruments the IP protocol
<table>
<thead>
<tr>
<th>Name</th>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vnc</td>
<td>Xvnc4viewer 4.1.1</td>
<td>VNC remote display client</td>
</tr>
<tr>
<td>irc</td>
<td>BitchX 1.1-4</td>
<td>IRC client</td>
</tr>
<tr>
<td>ftp</td>
<td>FTP 0.17-18</td>
<td>FTP client</td>
</tr>
<tr>
<td>ssh</td>
<td>SSH 1.5.1p1</td>
<td>Secure shell</td>
</tr>
<tr>
<td>vlc</td>
<td>VLC 0.8.6</td>
<td>VLC media player client</td>
</tr>
<tr>
<td>web</td>
<td>Firefox 3.5.3</td>
<td>Firefox web browser</td>
</tr>
<tr>
<td>mail</td>
<td>Thunderbird 3.0.1</td>
<td>Thunderbird mail client</td>
</tr>
</tbody>
</table>

Table 9.2 – Application scenarios

stack at well-defined points during the traversal of the stack by a packet, and provides hooks to execute desired code at these points.

The measurements were conducted on an IBM HS20 eServer BladeCenter. Each blade had dual 3.06 GHz Intel Xeon\textsuperscript{T\textregistered} CPUs, 2.5 GB RAM, a 40 GB local disk, and Q-Logic Fibre Channel 2312 host bus adapters. The blades were interconnected with a Gigabit Ethernet switch and connected through Fibre Channel to an IBM FastT500 SAN controller with an Exp500 storage unit with ten 70 GB IBM Fibre hard drives. Each blade used the GFS cluster file system [155] to access the shared SAN.

We used the applications listed in Table 9.2. We ran each application alone within a user’s standard desktop environment. To do so, we populated a container with a TightVNC server that executed the application. For applications that are non-GUI programs such as ftp, we first ran an xterm window, and then launched the program in it. In all the experiments we used three hosts to represent the user’s desktop, the NCS, and the remote service. We refer to them as host, NCS, and remote, respectively. Both the host and the NCS resided in the same physical subnet with access to a shared file system. The remote host resided on a different subnet.

We first quantify the migration of networked containers in terms of migration time and the amount of state transferred. To measure these properties we migrated each application from the host to the NCS and back, and averaged the results over
ten iterations. Each individual migration consisted of a taking a checkpoint of the applications on the origin host, streaming the checkpoint image to the destination host and then restarting the application there. Each checkpoint was a full checkpoint (i.e., non-incremental) with deferred write. File system snapshots (and pre-snapshots) were unnecessary as the shared file system available on both hosts remained untouched during the migration. Pre-quiesce was also omitted as the goal was to minimize the total checkpoint time rather than the application downtime.

Figure 9.4 shows the average amount of data in megabytes transferred between the host and NCS for all the application scenarios. Nearly the entire migrated state accounted for the container and applications stand-alone state. The amount of network data was negligible in all cases and could not be plotted on the graph. The amount of application state is modest, and varies from several megabytes to under 50 MB. As expected, the amount of state highly depends on the application that was migrated: heavier, GUI based applications such as Firefox and Thunderbird require more live state to be transferred. The vnc case serves as a base case since every networked container must contain at least one instance of a VNC server for the display virtualization. It provides an estimate of the memory footprint of an empty container. Subtracting this number from the amount of state transferred for any application provides an estimate of the memory footprint of that particular application.

Figure 9.5 shows the averages migration time in milliseconds between the host and the NCS for all the application scenarios. The results demonstrate that NetCont can successfully migrate the network state and persistent connections of networked containers. Moreover, the total migration time for all application scenarios is below 600 ms, well under the time it takes the host to suspend or resume, which is about 2 seconds each on our test machines. In other words, NetCont can evacuate the host fast enough and without slowing down the host’s suspend operation.
Figure 9.5 also provides a breakdown of the migration time into three components: the *prepare* time is the time to contact the destination computer and create a container there with a set of processes for the incoming applications; the *network* time indicates how long it takes to reconstruct the network state (including both the host network state and the network connections state) at the destination; lastly, *transfer* indicates the time to transfer the entire stand-alone container and applications state.

The total migration times correlate well with the total amount of state transferred, and is limited by the network bandwidth. The time to prepare for migration is similar for all application scenarios. The time to restore the network connections is negligible compared to the total migration time, which is dominated by the transfer time of the applications state. The time for the data transfer to complete is when the origin of the migration becomes free, and is under 400 ms in all cases, which is lower than the total migration time. Observe that when migrating out, the host is liberated earlier.
than when bringing back the applications, as the latter must also include the time spent restoring the applications state after that data transfer completes.

To evaluate the effectiveness of NetCont in yielding meaningful energy savings, we measured its power consumption in several configurations: Regular Host for a regular desktop not running NetCont, NetCont Host for a desktop with NetCont, NCS Idle for an idle NCS (i.e., without containers), NCL Busy for an NCS loaded with twenty containers (we also ensured that its entire memory is allocated and in use), and Suspended Host for a desktop in sleep state. For the measurements we used a Kill-A-Watt P4400 power meter, and recorded the consumption in idle state and in busy state.

The results for power consumption measurements for all configurations are given in Table 9.3. The results show that the power usage during idle times is 95–96 W in all cases, except, of course, the suspend case. The power consumption in busy times was
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Host</td>
<td>Desktop, X session</td>
<td>95 W</td>
</tr>
<tr>
<td>NetCont Host</td>
<td>Desktop, X session, a container</td>
<td>96 W</td>
</tr>
<tr>
<td>NCS (empty)</td>
<td>NCS (no containers)</td>
<td>95 W</td>
</tr>
<tr>
<td>NCS (in use)</td>
<td>NCS with 20 containers</td>
<td>96 W</td>
</tr>
<tr>
<td>Suspended Host</td>
<td>Desktop in suspend state</td>
<td>2 W</td>
</tr>
</tbody>
</table>

Table 9.3 – NetCont power consumption

around 110–130 W most of the time, but could reach 150 W. The results demonstrate that the use of virtualization for the applications (with containers), the display, or the file system does not impact the hosts’ power consumption. Nor does the number of containers staged on the NCS or its memory usage appear to impact the power drain. These findings suggest that the scalability of the system is not determined by power consumption of idle containers, but rather by the CPU, network, and particularly memory resources available on the NCS.

Based on the session memory measurements in Figure 9.4, we project that the power load per session on a high end server being used as an NCS will be minimal. We conservatively estimate that each migrated desktop session will, on average, fit within under 64 MB footprint as this is significantly more than the total of the applications listed in Figure 9.4. This means that a server can accommodate at least 16 sessions per GB or RAM installed on it. Therefore a dual Intel quad core 5570 with 32 GB, which consumes only a little over 300 W when under load [67], can conveniently hold networked applications from at least 512 desktops. This means that each session’s amortized power consumption will be less than 1 W. This is equivalent to the host’s power drain when suspended, and not only significantly less than the idle power usage, but also comparable to the consumption of the specialized embedded hardware used by Somniloquy [5].
9.5 Summary

In this chapter, we presented NetCont, a system to reduce power usage by enabling mobility of individual desktop applications. Instead of letting hosts sit idle when they are not being actively used, and thereby wasting lots of power, active networked applications can be moved to a Network Connection Server (NCS) that lets unmodified applications continue to run exactly as if they had remained on the original host. By consolidating applications from many individual idle machines onto a single NCS, NetCont can enable significant energy saving across the enterprise. The key innovation that makes this possible is the use of transparent checkpoint-restart to migrate the applications and then leverage their built in know-how, instead of having to rewrite applications and figure out how to separate their network functionalities. Our experimental results demonstrate that NetCont can significantly reduce the power usage of idle desktops, by enabling them to go into a deep sleep state that cuts their power usage by 98%. Migration of networked applications to and from the NCS occurs in sub-second times, significantly less than the time it takes a machine to suspend and resume, and therefore does not impact the availability of services on the machine.
Chapter 10

Checkpoint-Restart in Linux

In Chapter 6 we argued in favor of placing checkpoint-restart functionality in the operating system kernel to transparently support unmodified applications, which is crucial in practice to enable deployment and widespread use. Given its benefits, the kernel level approach was also adopted by several other projects that aim to provide application checkpoint-restart for Linux [48, 52, 95, 121, 124, 188]. However, none of them is integrated into the mainstream Linux; instead, they are all implemented as kernel modules or sets of kernel patches instead. As such, they incur a burden on both users because they are cumbersome to install, and developers because maintaining them on top of quickly changing upstream kernels is a sisyphean task and development quickly falls behind. In this chapter, we build on the experience garnered with Deja-View to introduce an alternative kernel checkpoint-restart implementation aimed for inclusion in the mainline Linux kernel.

We present Linux-CR, an in-kernel implementation of transparent application checkpoint-restart that is transparent, secure, reliable, and efficient, and does not adversely impact the performance or code quality of the rest of the Linux kernel. Linux-CR benefits from several supporting features needed for checkpoint and restart
that are already available in the mainline Linux kernel, including the ability to isolate applications inside containers, and to selectively freeze applications. Because Linux-CR is purposed for a mainstream operating system, we focus on several important practical aspects. We describe Linux-CR’s usage model from a user’s point of view and the userspace tools and APIs available for users. We introduce a novel mechanism to test whether a checkpointed container is correctly isolated, providing a means to ensure the restartability of checkpoints, which is critical for real-world adoption. We discuss how Linux-CR handles errors while providing sufficient details about them for callers to determine the root cause, and how Linux-CR addresses security concerns associated with complex and powerful tools such as checkpoint and restart. Finally, we present Linux-CR’s abstractions in the kernel and the corresponding kernel API from a developer’s point of view.

10.1 Usage

The granularity of checkpoint-restart in Linux-CR is a process hierarchy. A checkpoint begins at a task which is the root of the hierarchy, and proceeds recursively to include all the descendant processes. A checkpoint generates an image that represents the state of the process hierarchy. A restart takes a checkpoint image as input to create an equivalent process hierarchy and restore the state of the processes accordingly.

Before a checkpoint begins, and for the duration of the entire checkpoint, all processes in the hierarchy must be frozen. Like in DejaView, this is necessary to prevent them from modifying system state while a checkpoint is underway, and thus avoid inconsistencies from occurring in the checkpoint image. Freezing the processes also puts them in a known state—just before returning to userspace—which is useful because it is a state with only a trivial kernel stack to save and restore.
Linux-CR supports two main forms of checkpoint: *container checkpoint* and *sub-tree checkpoint*. The distinction between them depends on whether the checkpointed hierarchy is “self contained” and “isolated”. The term “self contained” refers to a hierarchy that includes all the processes that are referenced in it. In particular, it must include all parent, child, and sibling processes, and also orphan processes that were re-parented. The term “isolated” refers to a hierarchy whose resources are only referenced by processes that belong to the hierarchy. For example, open file handles held by processes in the hierarchy may not be shared by processes not in the hierarchy. A key property of hierarchies that satisfy both conditions is that their checkpoints are consistent and reliable, and therefore also *restartable*.

*Container checkpoint* operates on process hierarchies that are both isolated and self contained. In Linux, isolated and self-contained hierarchies can be created using *namespaces* [19], which facilitate the provision of private sets of resources for groups of processes. In particular, the PID-namespace can be used to generate a sub-hierarchy that is self contained. A useful management tool for this is Linux Containers [109], which leverages namespaces to encapsulate applications inside virtual execution environments to give them the illusion of running privately.

Checkpointing an entire container ensures that processes inside the container do not depend on processes from the outside. To ensure also that the checkpoint is consistent, the state of shared resources in use by processes in the container must remain unmodified for the duration the checkpoint. Because the processes in the container are frozen they may not alter the state. Thus it suffices to require that, at checkpoint time, none of the resources are referenced by processes from the outside. Combined, these properties guarantee that a future restart from a container checkpoint will always succeed. Note that to leverage container checkpoint, users must launch those applications that they wish to checkpoint inside containers.
Subtree checkpoint operates on arbitrary hierarchies. Subtree checkpoints are especially handy since they do not require users to launch their applications in a specific way. For example, casual users that execute long-running computations can simply checkpoint their jobs periodically. However, subtree checkpoints cannot provide the same guarantees as container checkpoints. For instance, because checkpoint iterates the hierarchy from the top down, it will not reach orphan processes unless it begins with the init(1) process, so orphan processes will not be recreated and restored at restart. Instead, it is the user’s responsibility to ensure that dependencies on processes outside the hierarchy and resource sharing from outside the hierarchy either do not exist or can be safely ignored. For example, a parent process may remain outside the hierarchy if we know that the child process will never attempt to access it. In addition, if outside processes share state with the container, they must not modify that state while checkpoint takes place.

Linux-CR supports a third form of checkpoint: self-checkpoint. Self-checkpoint allows a running process to checkpoint itself and save its own state, so that it can restart from that state at a later time. It does not capture the relationships of the process with other processes, or any sharing of resources. It is most useful for standalone processes wishing to be able to save and restore their state. Self-checkpoint occurs by an explicit system call and always takes place in the context of the calling process. The process need not be frozen for the duration of the checkpoint. This form of checkpoint is analogous to the fork system call in that the system call may return in two different contexts: one when the checkpoint completes and another following a successful restart. The process can use the return value from the system call to distinguish between the two cases: a successful checkpoint operation will return a “checkpoint ID” which is a non-zero positive integer, while a successful restart operation always returns zero.
10.1.1 Userspace Tools

The userspace tools consist of three programs: checkpoint to take a checkpoint of a process hierarchy, restart to restart a process hierarchy from a checkpoint image, and ckptinfo to provide information on the contents of a checkpoint image. A fourth utility, nsexec, allows users to execute applications inside a container by creating an isolated namespace environment for them. These tools provide the most basic userspace layer on top of the bare kernel functionality. Higher level abstractions for container management and related functionality are provided by packages like lxc [108] (of Linux Containers) and libvirt [107].

Figure 10.1 provides a practical example that illustrates the steps involved to launch an application inside a container, then checkpoint it, and finally restart it in another container. In the example, the user launches a session with an sshd daemon and a screen server inside a new container.

Lines 1–3 create a freezer control group to encapsulate the process hierarchy to be checkpointed. Lines 5–13 create a script to start the processes, and the script is executed in line 15. In line 7 the script joins the freezer control group. Descendant processes will also belong there by inheritance, so that later it will be possible to freeze the entire process hierarchy. Lines 8–10 close the standard input, output and error streams to decouple the new container from the current environment and ensure that it does not depend on tty devices.

In line 15, nsexec launches the script inside a new private set of namespaces. Lines 18–21 show how the application is checkpointed. First, in line 18 we freeze the processes in the container using the freezer control group. We use the checkpoint utility in line 19 to checkpoint the container, using the script’s PID to indicate the root of the target process hierarchy. In the example, we kill the application once the
checkpoint is completed, and thaw the (now empty) control group. We restart the application in line 23. Finally, in line 25 we show how to examine the contents of the checkpoint image using the `ckptinfo` utility.

The log files provided to both the `checkpoint` and `restart` commands are used for status and error reports. Should a failure occur during either operation, the log file will contain error messages from the kernel that carry more detailed information about the nature of the error. Further debugging information can be gained from the checkpoint image itself with the `ckptinfo` command, and from the restart program by using the “-vd” switches.
### Table 10.1 – System call flags (checkpoint and restart)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Flags</th>
<th>Flag description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoint</td>
<td>CHECKPOINT_SUBTREE</td>
<td>perform a subtree checkpoint</td>
</tr>
<tr>
<td></td>
<td>CHECKPOINT_NETNS</td>
<td>include network namespace state</td>
</tr>
<tr>
<td>Restart</td>
<td>RESTART_TASKSELF</td>
<td>perform a self-restart</td>
</tr>
<tr>
<td></td>
<td>RESTART_FROZEN</td>
<td>freeze all the restored tasks after restart</td>
</tr>
<tr>
<td></td>
<td>RESTART_GHOST</td>
<td>indicate a process that is a place-holder</td>
</tr>
<tr>
<td></td>
<td>RESTART_KEEP_LSM</td>
<td>restore saved MAC labels (if permitted)</td>
</tr>
<tr>
<td></td>
<td>RESTART_CONN_RESET</td>
<td>force open sockets to a closed state</td>
</tr>
</tbody>
</table>

### 10.1.2 System Calls

The userspace API consists of two new system calls for checkpoint and restart:

- `long checkpoint(pid, fd, flags, logfd)`

This system call serves to request a checkpoint of a process hierarchy whose root task is identified by `@pid`, and pass the output to the open file indicated by the file descriptor `@fd`. If `@logfd` is not `-1`, it indicates an open file to which error and debug messages are written. Finally, `@flags` determines how the checkpoint is taken, and may hold one or more of the values listed in Table 10.1.

On success the system call returns a positive checkpoint identifier. In the future, a checkpoint image may optionally be briefly preserved in kernel memory. The identifier would serve to reference that image. On failure the system call returns a suitable negative error value. In self-checkpoint, where a process checkpoints itself, it is necessary to distinguish between the first return from a successful checkpoint, and a subsequent return from the same system call but following a successful restart. This distinction is achieved using the return value, similarly to `fork`. Specifically, the system call returns plain 0 if it came from a successful restart. In either case, when the operation fails the return value is a suitable negative error value.
• `long restart(pid, fd, flags, logfd)`

This system call serves to restore a process hierarchy from a checkpoint image stored in the open file indicated by the file descriptor `fd`. It is intended to be called by all the restarting tasks in the hierarchy, and by a special process that coordinates the restart operation. When called by the coordinator, `pid` indicates the root task of the hierarchy (as seen in the coordinator’s PID-namespace). The root task must be a child of the coordinator. When not called by the coordinator, `pid` must remain 0. If `logfd` is not -1, it indicates an open file to which error and debug messages are written. Finally, `flags` determines how the checkpoint is taken, and may hold one or more of the values listed in Table 10.1.

On success the system call returns in the context of the process as it was saved at the time of the checkpoint. The exact behavior depends on how and when the checkpoint was taken. If the process was executing in userspace prior to the checkpoint, then the restart will arrange for it to resume execution in userspace exactly where it was interrupted. If the process was executing a system call, then the return value will be set to the return value of that system call whether it completed or was interrupted. For a self-checkpoint, the restart will arrange for the process to resume execution at the first instruction after the original call to `checkpoint`, with the system call's return value set to 0 to indicate that this is the result of a restart. On failure the return value is a suitable negative error value.

### 10.2 Architecture

The crux of checkpoint-restart is a mechanism to serialize the execution state of a process hierarchy, and to restore the process hierarchy and its state from the saved
state. For checkpoint-restart of multi-process applications, not only must the state associated with each process be saved and restored, but the state saved and restored must be globally consistent and preserve process dependencies. Furthermore, for checkpoint-restart to be useful in practice, it is crucial that it transparently support existing applications.

To guarantee that the state saved is globally consistent among all processes in a hierarchy, we must satisfy two requirements. First, the processes must be frozen for the duration of the checkpoint. Second, the resources that they use must not be modified by processes not in the hierarchy. These requirements ensure that the state will not be modified by processes in or outside the hierarchy while the execution state is being saved.

To guarantee that the operation is transparent to applications, we must satisfy two more requirements. First, the state must include all the resources in use by processes. Second, resource identifiers in use at the time of the checkpoint must be available when the state is restored. These requirements ensure not only that all the necessary state exists when restart completes, but also that it is visible to the application as it was before the checkpoint, so that the application remains unaware.

Linux-CR builds on the freezer subsystem to achieve quiescence of processes. This subsystem was created to allow the kernel to freeze all userspace processes in preparation for a full system suspend to disk. To accommodate checkpoint-restart, the freezer subsystem was recently re-purposed to enable freezing of groups of processes, with the introduction of the freezer control group. Freezing processes with the freezer control group is a simpler and more maintainable analog to DejaView’s notion of quiescing processes by overloading the semantics of SIGSTOP.

Linux-CR leverages namespaces [19] to encapsulate processes in a self-contained unit that isolates them from other processes in the system and decouples them from
the underlying host. Namespaces provide virtual private resource names: resource identifiers within a namespace are localized to the namespace. Not only are they invisible to processes outside that namespace, but they do not collide with resource identifiers in other namespaces. Thus, resource identifiers, such as PIDs, can remain constant throughout the life of a process even if the process is checkpointed and later restarted, possibly on a different machine. Without it, identifiers may in fact be in use by other processes in the system. To accommodate checkpoint-restart, namespaces were extended to allow restarting processes to select predetermined identifiers upon the allocation of their resources, so that those processes can reclaim the same set of identifiers they had used prior to the checkpoint.

For simplicity, we describe the checkpoint-restart mechanism assuming container checkpoint. We also assume a shared storage (across participating machines), and that the file system remains unmodified between checkpoint and restart. In this case, the file system state is not generally saved and restored as part of the checkpoint image, to reduce checkpoint image size. As before, available file system snapshot functionality [24, 53, 117] can be used to also provide a checkpointed file system image. We focus only on checkpointing process state; details on how to checkpoint file system, network, and device state are beyond the scope of this chapter.

10.2.1 Kernel vs. Userspace

Previous approaches to checkpoint-restart have run the gamut from fully in-kernel to hybrid to fully userspace implementations. Many properties of processes and resources can be recorded and restored in userspace [34, 40]. Ckpt [34] modifies tasks to let them checkpoint themselves. The checkpoint images are in the form of executable files which, when executed, restart the original process. CryoPID [40] also uses an
executable file for the checkpoint image, but relies on /proc information to checkpoint a task. However, neither is able to capture or restore some parts of a process’s system state, and both are limited in which applications they support. Some state exists that cannot be recorded or restarted from userspace.

To provide application transparency and allow applications to use the full range of operating system services, we chose to implement checkpoint-restart in the kernel. In addition, an in-kernel implementation is not limited to user visible APIs such as system calls, but rather it can use the full range of kernel APIs. This not only simplifies the implementation, but also allows use of native locking mechanisms to ensure atomicity at the desired granularity.

Checkpoint is performed entirely in the kernel. Restart is also done in the kernel, however, for simplicity and flexibility, the creation of the process hierarchy is done in userspace. Moving some portion of the restart to userspace is an exception, which is permitted under two conditions: first, it must be straightforward and leverage existing userspace APIs (i.e., not introduce specialized APIs). Second, doing so in userspace should bring significant added value, such as improved flexibility. Also, all userspace work must occur before entering the kernel, to avoid transitions in and out of the kernel.

For instance, the incentive to do process creation in userspace is because it is simple to use the clone system call to do so, and because it allows for great flexibility for restarting processes to do useful work after the process hierarchy is created and before the rest of the restart takes place. Indeed, the entire hierarchy is created before in-kernel restart is performed. Likewise, it is desirable to restore network namespaces in userspace. ¹ Doing so will allow reuse of existing userspace network setup tools.

¹However, as of the writing of this dissertation, this is yet undecided.
that are well understood instead of replicating their high-level functionality inside the kernel. Moreover, it will allow users to easily adjust network settings at restart time, e.g., change the network device or its setup, or add a firewall to the configuration.

10.2.2 Checkpoint and Restart

A checkpoint is performed in the following steps (steps 2–4 are done by the \texttt{checkpoint} system call in the kernel):

1. Freeze the process hierarchy to ensure that the checkpoint is globally consistent.

2. Record global data, including configuration and state that are global to the container.

3. Record the process hierarchy as a list of all checkpointed processes, their PIDs, and relationships.

4. Record the state of individual processes, including credentials, blocked and pending signals, CPU registers, open files, virtual memory, etc.

5. Thaw the processes to allow them to continue executing, or terminate the processes in case of migration. (If a file system snapshot is desired, it is taken prior to this step.)

Checkpoint is done by an auxiliary process, and does not require the collaboration of processes being checkpointed. This is important since processes that are not runnable, e.g., stopped or traced, would not be able to perform their own checkpoint. Moreover, this can be extended in the future to multiple auxiliary processes for faster checkpoint times of large process hierarchies.
Much effort was put to make checkpoint robust in the sense that if a checkpoint succeeds then, given a suitable environment, restart will succeed too. The implementation goes to great extents to be able to detect whether checkpointed processes are “non-restartable”. This can happen, for example, when a process uses a resource that is unsupported for checkpoint-restart, therefore it will not be saved at all. Even if a resource is supported, it may be temporarily in an unsupported state. For example, a socket that is in the process of establishing a connection is currently unsupported.

A restart is performed in the following steps (step 3 is done by the \texttt{restart} system call in the kernel):

1. Create a new container for the process hierarchy, and restore its configuration and state.

2. Create the process hierarchy as prescribed in the checkpoint image.

3. Restore the state of individual processes in the same order as they were checkpointed.

4. Allow the processes to continue execution (It is also possible to freeze the restarted processes, which is useful for debugging, for example.)

Restart is managed by a special \textit{coordinator} process, which supervises the operation but is not a part of the restarted process hierarchy. The coordinator process creates and configures a new container, and then generates the new process hierarchy in it. Once the hierarchy is ready, all the processes execute a system call to complete the restart of each process in-kernel.

To produce the process hierarchy, it is necessary to preserve process dependencies, such as parent-child relationships, threads, process groups, and sessions. The restored hierarchy must satisfy the same constraints imposed by process dependencies
at checkpoint. Because the process hierarchy is constructed in userspace, these dependencies must be established at process creation time (to leverage the existing system call semantics). The order in which processes are created is important, because some dependencies are not reflected directly from the hierarchical structure. For instance, an orphan process must be recreated by a process that belongs to the correct session group to correctly inherit that group. Linux-CR leverages the DumpForest and MakeForest algorithms presented in §6.5 to reconstruct a process hierarchy that is equivalent to the original one at the time of checkpoint.

In rebuilding the process hierarchy, there are two special cases of PIDs referring to terminated processes that require additional attention. One case is when a PID of a dead process is used as a PGID of another process. In this case, the restart algorithm creates a “ghost” process that serves as a place-holder that lives long enough so that its PID can be used as the PGID of another process, but terminates once the restart completes (and before the hierarchy may resume its execution, to avoid races). Another case is when a PID represents a zombie process that has exited but whose state has not been cleaned up yet. In this case, the restart algorithm creates a process, restores only minimal state such as its exit code, and finally the process exits to become a zombie.

Because the process hierarchy is created in userspace, the restarting processes have the flexibility to do useful work before eventually proceeding with in-kernel restart. For instance, they might wish to create a new custom networking route or filtering rule, create a virtual device which existed at the host at the time of checkpoint, or massage the mounts tree to mask changes since checkpoint.

Once the process hierarchy is created, all the processes invoke the restart system call and the remainder of the restart takes place in the kernel. Restart is done in the same order that processes were checkpointed. The restarting processes now wait for
their turn to restore their own state, while the coordinator orchestrates the restart.

Restart is done within the context of the process that is restarted. Doing so allows us to leverage the available kernel functionality that can only be invoked from within that context. Unlike checkpoint, which requires observing process state, restart is more complicated as it must create the necessary resources and reinstate their desired state. Being able to run in process context and leverage available kernel functionality to perform these operations during restart significantly simplifies the restart mechanism.

In the kernel, the restart system call depends on the caller. The coordinator first creates a common restart context data structure to share with all the restarting processes, and waits for them to become properly initialized. It then notifies the first process to begin the restart, and waits for all the restarting tasks to finish. Finally, the coordinator notifies the restarting tasks to resume normal execution, and then returns from the system call.

Correspondingly, restarting processes first wait for a notification from the coordinator that indicates that the restart context is ready, and then initialize their state. Then, each process waits for its turn to run, restores the state from the checkpoint image, notifies the next restarting process to run, and waits for another signal from the coordinator indicating that it may resume normal execution. Thus, processes may only resume execution after all the processes have successfully restored their state (or fail if an error has occurred), to prevent processes from returning to userspace prematurely before the entire restart completes.
10.2.3 The Checkpoint Image

The checkpoint image is an opaque *blob* (Binary Large OBject) of data, which is generated by the *checkpoint* system call and consumed by the *restart* system call. The blob contains data that describes the state of select portions of kernel structures, as well as process execution state such as CPU registers and memory contents. The image format is expected to evolve over time as more features are supported, or as existing features change in the kernel and require to adjust their representation. Any changes in the blob’s format between kernel revisions will be addressed by userspace conversion tools, rather than attempting to maintain backward compatibility inside the *restart* system call.

Internally, the blob consists of a sequence of records that correspond to relevant kernel data structures and represent their state. For example, there are records for process data, memory layout, open files, pending signals, to name a few. Each record in the image consists of a header that describes the type and the length of the record, followed by a payload that depends on the record type. This format allows userspace tools to easily parse and skim through the image without requiring intimate knowledge of the data. Keeping the data in self-contained records will also be suitable for parallel checkpointing in the future, where multiple threads may interleave data from multiple processes into a single stream.

Records do not simply duplicate the native format of the respective kernel data structures. Instead, they provide a representation of the state by copying those individual elements that are important. One justification is that during restart, one already needs to inspect, validate and restore individual input elements before copying them into kernel data structures. However, the approach offers three additional benefits. First, it improves image format compatibility across kernel revisions, being
agnostic to data structure changes such as reordering of elements, addition or deletion of elements that are unimportant for checkpoint-restart, or even moving elements to other data structures. Second, it reduces the total amount of state saved since many elements may be safely ignored. Per-process variables that keep scheduler state are one such example. Third, it allows a unified format for architectures that support both 32-bit and 64-bit execution, which simplifies process migration between them.

The checkpoint image is organized in five sections: a header, followed by global data, process hierarchy, the state of individual processes, and a finally a trailer. The header includes a magic number (to identify the blob as a checkpoint image), an architecture identifier in little-endian format, a version number, and some information about the kernel configuration. It also saves the time of the checkpoint and the flags given to the system call. It is followed by an architecture dependent header that describes hardware specific capabilities and configuration.

The global data section describes configuration and state that are global to the container being checkpointed. Examples include Linux Security Modules (LSM) and network devices and filters. In the future container-wide mounts may also go here. The process hierarchy section that follows provides the list of all checkpointed processes, their PIDs and their relationships, e.g., parent-child, siblings, threads, and zombies. These two sections are strategically placed early in the image for two reasons: first, it allows restart to create a suitable environment for the rest of the restart early on, and second, it allows to do so in userspace.

The remainder of the checkpoint image contains the state of all of the tasks and the shared resources, in the order that they were reached by the process hierarchy traversal. For each task, this includes state like the task structure, namespaces, open files, memory layout, memory contents, CPU state, signals and signal handlers, etc. Finally, the trailer that concludes the entire image serves as a sanity check.
The checkpoint-restart logic is designed for streaming to support operation using a sequential access device. Process state is saved during checkpoint in the order in which it needs to be used during restart. An important benefit of this design is that the checkpoint image can be directly streamed from one machine to another across the network and then restarted, to accomplish process migration. Using a streaming model provides the ability to pass checkpoint data through filters, resulting in a flexible and extensible architecture. Example filters include encryption, signature and validation, compression, and conversion between formats of different kernel versions.

### 10.2.4 Shared Resources

Shared resources may be referenced multiple times, e.g., by multiple processes or even by other resources. Examples of resources that may be shared include files descriptors, memory address spaces, signal handlers and namespaces. During checkpoint, shared resources will be considered several times as the process hierarchy is traversed, but their state need only be saved once.

To ensure that shared resources are saved exactly once, we need to be able to uniquely identify each resource, and keep track of resources that have been saved already. More specifically, when a resource is first discovered, it is assigned a unique identifier (tag) and registered in a hash-table using its kernel address (at checkpoint) or its tag (at restart) as a key. The hash-table is consulted to decide whether a given resource is a new instance or merely a reference to one already registered. Note that the hash-table itself is not saved as part of the checkpoint image; instead, it is rebuilt dynamically during both checkpoint and restart, and discarded when they complete.

During checkpoint, shared resources are examined by looking up their kernel addresses in the hash-table. If an entry is not found, then it is a new resource— we assign
a new tag and add it to the hash-table, and then record its state. Otherwise, the resource has been saved before, so it suffices to save only the tag for later reference.

During restart the state is restored in the same order as has been saved originally, ensuring that the first appearance of each resource is accompanied with its actual recorded state. As with checkpoint, saved resource tags are examined by looking them up in the hash-table, and if not found, we create a new instance of the required resource, restore its state from the checkpoint image, and add it to the hash-table. If an entry is found, it points to the corresponding (already restored) resource instance, which is reused instead of creating a new one.

10.2.5 Leak Detection

In order to guarantee that a container checkpoint is consistent and restartable, we must ensure that the container is isolated and that its resources are not referenced by other processes from outside the container. When container shared resources are referenced from outside, we say that they leak; the ability to detect leaks is a prerequisite for container checkpoint to succeed.

Because the shared objects hash-table already tracks shared resources, it also plays a crucial role in detecting resource leaks that may obstruct a future restart. The key idea behind leak detection is to explicitly count the total number of references to each shared object inside the container, and compare them to the global reference counts maintained by the kernel for these objects. If for a certain resource the two counts differ, it must be because of an external (outside the container) reference.

Leak detection begins in a pre-pass that takes place prior to the actual checkpoint, to ensure that there are no external references to container shared objects. In this pass we traverse the process hierarchy like in the actual checkpoint but do not save
the state. Instead, we collect the shared resources into the hash-table and maintain their reference counts. When this phase ends, the reference count of each object in the hash-table reflects the number of in-container references to it. We now iterate through all the objects and compare that count to the one maintained by the kernel. The two counts match if and only if there are no external references. (To be precise, the hash-table count should be one less, because it does not count the reference that the hash-table itself takes.)

Note that this procedure is non-atomic in the sense that the reference counts from the hash-table and the kernel are compared only after all the resources have been collected. This is racy because processes outside the container may modify the state of shared resources, create new ones or destroy existing ones before the procedure concludes. For instance, consider two processes, one inside a container and the other not, that share a single file descriptor table. Suppose that after the pre-pass collects the file table and the files in it, the outside process opens a new file, closes another (existing) file, and then terminates. At this point, the new file is left out of the hash-table, while the other (closed) file remains there unnecessarily. Moreover, when the pre-pass concludes it will not detect a file table leak because the outside process exited, and the file table is only referenced inside the container. It will not detect a file leak even though the new file may be referenced outside, because the new file had not even been tracked.

To address these races we employ additional logic for leak detection during the actual checkpoint\(^2\). This logic can detect in-container resources that are not tracked by the hash-table, or that are tracked but are no longer referenced in the container. Untracked resources are easy to detect, because their lookup in the hash-table will

\(^2\)Leak detection was first proposed by OpenVZ [121], but their logic does not address these races thereby allowing untracked and deleted resources to remain undetected.
fail. To detect deleted resources, we mark every resource in the hash-table that we save during the checkpoint, and then at the end of the checkpoint, we verify that all the tracked resources are marked. A tracked but unmarked resource must have been added to the hash-table and then deleted before being reached by the actual checkpoint. In either case, the checkpoint is aborted.

10.2.6 Error Handling

Both checkpoint and restart operations may fail due to a variety of reasons. When a failure does occur, they must provide proper cleanup. For checkpoint this is simple, because the checkpoint operation is non-intrusive: the process hierarchy whose state is saved remains unaffected. For restart, cleanup is performed by the coordinator, which already keeps track of all processes in the restored hierarchy. More specifically, in case of failure the coordinator will send a fatal signal to terminate all the processes before the system call returns. Because the cleanup is part of the return path from the **restart** system call exit path, there is no risk that cleanup be skipped should the coordinator itself crash.

When a checkpoint or restart fails, it is desirable to communicate enough details about the failure details to the caller to determine the root cause. Using a simple, single return value from the system call is insufficient to report the reason of a failure. For instance, a process that is not frozen, a process that is traced, an outstanding asynchronous IO transfer, and leakage of shared resource leakage, are just a few failure modes that result all in the error **-EBUSY**.

To address the need to report detailed information about a failure, both **checkpoint** and **restart** system calls accept an additional argument: a file descriptor to which the kernel can write diagnostic and debugging information. Both the checkpoint and
restart userspace utilities have options to specify a filename to store this log.

In addition, checkpoint stores in the checkpoint image informative status report upon failure in the form of (one or more) error objects. An error object consists of a mandatory pre-header followed by a null character (‘\0’), and then a string that describes the error. By default, if an error occurs, this will be the last object written to the checkpoint image. When a failure occurs, the caller can examine the image and extract the detailed error message. The leading ‘\0’ is useful if one wants to seek back from the end of the checkpoint image, instead of parsing the entire image separately.

10.2.7 Security Considerations

The security implications of in-kernel checkpoint-restart require careful attention. A key concern is whether the system calls should require privileged or unprivileged operation. Originally our implementation required users to have the CAP_SYS_ADMIN capability, while we optimistically asserted our intent to eventually remove the need for privilege and allow all users to safely use checkpoint and restart. However, it was pointed out that letting unprivileged users use these system calls is not only beneficial to users, but also has the useful side effect of forcing the checkpoint-restart developers to be more careful with respect to security throughout the design and development process. In fact, we believe this approach has succeeded in keeping us more on our toes and catching ways that users otherwise would have been able to escalate privileges through carefully manipulated checkpoint images, for instance bypassing CAP_KILL requirements by specifying arbitrary userid and signals for file owners.

At checkpoint, the main security concern is whether the process that takes a checkpoint of other processes in some hierarchy has sufficient privileges to access that
state. We address this by drawing an analogy between checkpointing and debugging processes: in both it is necessary for an auxiliary process to gain access to internal state of some target process(es). Therefore, for checkpoint we require that the caller of the system call will be privileged enough to trace and debug (using `ptrace`) all of the processes in the hierarchy.

For restart, the main concern is that we may allow an unprivileged user to feed the kernel with random data. To this end, the restart works in a way that does not skip the usual security checks. Process credentials, i.e., `UID`, `EUID`, and the security context (security contexts are part of LSM) currently come from the caller, not the checkpoint image. To restore credentials to values indicated in the checkpoint image, restarting processes use the standard kernel interface. Thus, the ability to modify one’s credentials is limited to one’s privilege level when beginning the restart.

Keeping the restart procedure to operate within the limits of the caller’s credentials means that scenarios consisting of privileged applications that reduce their privilege level cannot be supported. For instance, a “setuid” program that opened a protected log file and then dropped privileges will fail the restart, because the user will not have enough credentials to reopen the file. The only way to securely allow unprivileged users to restart such applications is to make the checkpoint image tamper-proof.

There are a few ways to ensure the a checkpoint image is authentic. One method is to make the userspace utilities privileged using “setuid” and use cryptographic signatures to validate checkpoint images. In particular, checkpoint will sign the image and restart will first verify the signature before restoring from it. For instance, TPM [72] can be used to sign the checkpoint image and produce a keyed hash using a sealed private key, and to refuse restart in the absence of the correct hash. Another method is to create an assured pipeline for the checkpoint image, from the invocation of the
checkpoint and restart system calls. Assured pipelines are precisely a target feature of SELinux, and could be implemented by using specialized domains for checkpoint and restart. Note, however, that even with a tamper-proof checkpoint image, a concern remains that the checkpoint image amounts to a persistent privileged token, which a clever user could find ways to exploit in new and interesting ways.

### 10.3 Kernel Internal API

The kernel API consists of a set of functions for use in kernel subsystems and modules to provide support for checkpoint and restart of the state that they manage. All the kernel API calls accept a pointer to a checkpoint context (`ctx`), that identifies the operation in progress.

The kernel API can be divided by purpose into several groups: functions to handle data records; functions to read and write checkpoint images; functions to output debugging or error information; and functions to handle shared kernel objects and the hash-table. Table 10.2 lists the API groups and their naming conventions. Additional APIs exist to abstract away the details about checkpointing and restoring instances of some objects types, including memory objects, open files, and LSM (security) annotations.

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ckpt_hdr...</td>
<td>record handling (alloc/dealloc)</td>
</tr>
<tr>
<td>ckpt_write...</td>
<td>write data/objects to image</td>
</tr>
<tr>
<td>ckpt_read...</td>
<td>read data/objects from image</td>
</tr>
<tr>
<td>ckpt_msg</td>
<td>output to the log file</td>
</tr>
<tr>
<td>ckpt_err</td>
<td>report an error condition</td>
</tr>
<tr>
<td>ckpt_obj...</td>
<td>manage objects and hash-table</td>
</tr>
</tbody>
</table>

Table 10.2 – Kernel API by groups
The ckpt_hdr... group provides convenient helper functions to allocate and deallocate buffers used as intermediate store for the state data. During checkpoint they store the saved state before it is written out. During restart they store data read from the image before it is consumed to restore the corresponding kernel object.

The ckpt_write... and ckpt_read... groups provide helper functions to write data to and read data from the checkpoint image, respectively. These are wrappers that simplify the handling, for example by adding and removing record headers, and by providing shortcuts to handle common data such as strings and buffers.

The ckpt_msg function writes an error message to the log file (if provided by the user), and, when debugging is enabled, also to the system log. The ckpt_err function is used when checkpoint or restart cannot succeed. It accepts the error code to be returned to the user, and a formatted error message which is written to the user-provided log and the system log. If multiple errors occur, e.g., during restart, only the first error value will be reported, but all messages will be printed.

The ckpt_obj... group includes helper functions to handle shared kernel objects. They simplify the hash-table management by hiding details such as locking and memory management. They include functions to add objects to the hash-table, to find objects by their kernel address at checkpoint or by their tag at restart, and to mark objects that are saved (for leak detection).

Dealing with shared kernel objects aims to abstract the details of how to checkpoint and restart different object types, and how to push the code to do so near the native code for those objects. For example, code to checkpoint and restart open files and memory layouts appears in the file/ and mm/ subdirectories, respectively. The motivation for this is twofold. First, placing the checkpoint-restart code there improves maintainability because it makes the code more visible to maintainers. Higher awareness of maintainers increases the chances that they will adjust the checkpoint-
restart code when they introduce changes to other parts of the kernel code base. Second, it is more friendly to kernel objects that are implemented in kernel modules, because it means that the code to checkpoint-restart such objects is also part of the module.

To abstract the handling of shared kernel objects we associate a set of operations with each object type (similar to operations for files, sockets, etc). These include methods to checkpoint and restore the state of an object, methods to take or drop a reference to the objects so that it can be referenced when in the hash-table, and a method to read the reference count (in the hash-table) of an object. The function `register_checkpoint_obj` is used to register an operations set for an object type. It is typically called from kernel initialization code for the corresponding object, or from module initialization code as part of loading a new module. Each of the `ckpt_obj`... takes the object type as one of its argument, which indicates the object-specific set of operation to use.

During checkpoint, for each shared kernel object the function `checkpoint_obj` is called. It first looks up the object in the hash-table, and, if not found, invokes the `->checkpoint` method from the corresponding operations set to create a record for the object in the image, and adds the object to the hash-table. During restart, records of shared kernel objects in the input stream are passed to the function `restore_obj`, which invokes the `->restore` method from the corresponding operations set to create an instance of the object according to the saved state, and adds the newly created object to the hash-table.

Several objects in the Linux kernel are already abstracted using operations set, which contain methods describing how to handle different versions of objects. For instance, open files have `file_operations`, and seeking in `ext3` file system is performed using a different method than in `nfs` file system. Likewise, virtual memory areas have
vm_operations_struct, and the method to handle page faults is different in an area that corresponds to private anonymous memory than one that corresponds to shared mapped memory. For such objects, we extend the operations to also provide methods for checkpoint, restore, and object collection (for leak detection), as follows:

To checkpoint a virtual memory area in a memory address space of a task, the struct vm_operations_struct needs to provide the method for the ->checkpoint operation:

```
int checkpoint(ctx, vma)
```

and at restart, a matching callback to restore the state of a new virtual memory area object:

```
int restore(ctx, mm, vma_hdr)
```

Note that the function to restore cannot be part of the operations set, because it needs to be known before the object instance even exists.

To checkpoint an open file, the struct file_operations needs to provide the methods for the ->checkpoint and ->collect operations:

```
int checkpoint(ctx, file)
int collect(ctx, file)
```

and at restart, a matching callback to restore the state of an opened file:

```
int restore(ctx, file, file_hdr)
```

Here, too, the restore function cannot be part of the operations set. For most file systems, generic functions are sufficient: generic_file_checkpoint and generic_file_restore.

To checkpoint a socket, the struct proto_ops needs to provide the methods for the ->checkpoint, ->collect and ->restore operations:

```
int checkpoint(ctx, sock);
int collect(ctx, sock);
int restore(ctx, sock, sock_hdr)
```
10.4 Experimental Results

We evaluated the current version of Linux-CR to answer three questions: (a) how long checkpoint and restart take; (b) how much storage they require; and (c) how they scale with the number of processes and their memory size.

The measurements were conducted on a machine with two dual-core 2 GHz AMD Opteron processors with 1 GB L2 cache, 2 GB RAM, and a 73.4 GB 10025 RPM local disk. We used a Fedora 10 distribution with all optional system daemons on the test system turned off. We used a Linux 2.6.34 kernel with the Linux-CR v21 patch-set, with most debugging disabled and SELinux disabled.

For the measurements we used the makeprocs test from the checkpoint-restart test-suite [104]. This test program allows a caller to specify the number of child processes, a memory size for each task to map, or a memory size for each task to map and then dirty. We repeated the tests thirty times, and report average values.

Because Linux-CR aims for inclusion in the mainline Linux kernel, the current implementation has focused on correctness, cleanness, and coverage (i.e., the range of supported kernel resources), in this order. Advanced features such as performance optimizations were intentionally deferred for future work to reduce the code complexity in view of the public code review by the kernel developers community. In particular, the evaluated version does not include any of the optimizations mentioned in Chapter 6.

Table 10.3 presents the results in terms of checkpoint image size, checkpoint time and restart time, for measurements with 10, 100 and 1000 processes. Checkpoint times were measured once with the output saved to a local file, and once with the output discarded (technically, redirected to /dev/null). Restart times were measured from when the coordinator begins and until it returns from the kernel after a successful
Table 10.3 – Checkpoint-restart performance

<table>
<thead>
<tr>
<th>Number of processes</th>
<th>Image size</th>
<th>Checkpoint time (to file)</th>
<th>Checkpoint time (no I/O)</th>
<th>Restart time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.87 MB</td>
<td>8 ms</td>
<td>3 ms</td>
<td>10 ms</td>
</tr>
<tr>
<td>100</td>
<td>8.0 MB</td>
<td>72 ms</td>
<td>24 ms</td>
<td>72 ms</td>
</tr>
<tr>
<td>1000</td>
<td>79.6 MB</td>
<td>834 ms</td>
<td>237 ms</td>
<td>793 ms</td>
</tr>
</tbody>
</table>

operation. Restart times were measured reading the checkpoint images from a warm-cache.

The results show that the checkpoint image size and the time for checkpoint and restart scale linearly with the number of processes in the process hierarchy. The average total checkpoint time is about 0.8 ms per process when writing the data to the file system, and drops to well under 0.3 ms per process when file system access is skipped. Writing the data to a file triples the checkpoint time. This suggests that buffering the checkpoint output until the end of a checkpoint is a good candidate for a future optimization to reduce application downtime at checkpoint. Average total restart time from warm cache is about 0.9 ms per process. The total amount of state that is saved per process is also modest, though it highly depends on the applications being checkpointed.

To better understand how much of the checkpoint and restart times is spent for different resources, we instrumented the respective system calls to measure a breakdown of the total time. For both checkpoint and restart, saving and restoring the memory contents of processes amounted to over 80% of the total time. The total memory in use within a process hierarchy is also the most prominent component of the checkpoint image size.

To look more closely at the impact of the process size, we measured the checkpoint time for ten processes each with memory sizes increasing from 1 MB to 1 GB
Table 10.4 – Checkpoint times and memory sizes

<table>
<thead>
<tr>
<th>Task size</th>
<th>Checkpoint time (clean)</th>
<th>Checkpoint time (dirty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>0.5 ms</td>
<td>1.3 ms</td>
</tr>
<tr>
<td>10 MB</td>
<td>0.7 ms</td>
<td>6.4 ms</td>
</tr>
<tr>
<td>100 MB</td>
<td>1.7 ms</td>
<td>58 ms</td>
</tr>
<tr>
<td>1 GB</td>
<td>10.6 ms</td>
<td>337 ms</td>
</tr>
</tbody>
</table>

that the process allocated using the `mmap` system call. We repeated the test twice. In one instance the processes only allocate memory but do not touch it. In the second instance the processes also dirty all the allocated memory. In both cases, the output was redirected to avoid expensive I/O. The results are given in Table 10.4. The results show strong correlation between the memory footprint of processes and checkpoint times. Even when memory is untouched, the cost associated with scanning the process’s page tables is significant. Checkpoint times increase substantially with dirty memory as it requires the contents to actually be stored in the checkpoint image.

10.5 Summary

In this chapter, we presented in detail Linux-CR, an implementation of checkpoint-restart which aims for inclusion in the mainline Linux kernel. For several years the Linux kernel has been gaining the necessary groundwork for such functionality, and recently support for kernel based transparent checkpoint-restart is also maturing. The Linux-CR project emerged as a unifying framework to provide checkpoint-restart in the Linux kernel. It provides transparent, reliable, flexible, and efficient application checkpoint-restart. Aiming at a wide audience ranging from users to kernel developers, we also explained in detail the usage model and described the user interfaces and select key kernel interfaces. Previous work on application checkpoint and restart
for Linux did not address the crucial issue of integration with the mainstream Linux kernel. A key ingredient to a successful upstream implementation is the understanding by the kernel community of the usefulness of checkpoint-restart. That we have received much feedback shows that the usefulness of checkpoint-restart is recognized by the community. We have high hopes that, with the community’s help, the project will succeed in providing checkpoint-restart functionality in the upstream kernel.
Chapter 11

Related Work

This dissertation introduced a new personal virtual computer recorder model to the desktop. Its contributions touch upon several fields in systems, particularly regarding virtualization and checkpoint-restart. In this chapter we survey the prior art in these fields in detail and discuss their relationship to DejaView.

11.1 DejaView

DejaView is created in the spirit of Vannevar Bush’s Memex [25] 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 [64] is centered around digitally capturing a lifetime of Gordon Bell’s information with a focus on indexing and annotating individual documents. Lifestreams [60] 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 [43], Microsoft [44], Yahoo [45], and Apple [9], enable a user to search one’s desktop files. Connections [157] improves a user’s ability to search by extracting information that links which files were used with which programs and at which times to enable a user to perform a more contextualized search. The closest of these systems to DejaView is Stuff I’ve Seen (SIS) [49], 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 [182] 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 [10] 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.

Cohen et al. [36] present a method for automatically extracting from a running system an indexable signature that distills the essential characteristics from a system state. The signatures can then be subjected to automated clustering and similarity-based retrieval to identify when an observed system state is similar to a previously-observed state. While this method is related to DejaView at a high level, as they are both able to index past system state, Cohen et al.’s project is focused on identifying
when software acts in an unexpected way. They provide tools for understanding and
documenting the causes of the unexpected behavior and associating it with previous
system behavior to help resolving the problem. While this approach can work well for
services that one provides a customer, it is not easily applied to a dynamic desktop
of a user where things are constantly changing. Users are interested in finding and
retrieving their documents and computing state, not just correlating a desktop state
to a previous desktop state.

Screencasting provides a recording of a desktop’s screen that can be played back
at a later time [26, 170, 185]. Screencasting works by screenscraping and taking
screenshots of the display many times a second. There have also been VNC-based ap-
proaches to recording desktop sessions [100, 172], which incur less recording overhead
than taking frequent screenshots, but most of them are tailored towards improving
remote group collaboration. All of these require 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 over-
head, 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 [120] 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. However, using the VNC protocol may miss
intermediate data between sampled points. 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.
11.2 Content Recording

Screen readers are in common use today. As assistive technologies, they require access to GUI components in a standardized interface, which accessibility libraries provide. Modern solutions [83, 123] provide access to onscreen data, but are limited to watching changes in the foreground window only. Users can define hot spots, physical regions of the screen to scan for changes, but current screen readers lack the capability to detect more complicated changes, such as specific text strings appearing in arbitrary locations on the desktop. Also, requiring blind and partially sighted users to specify these physical locations can be cumbersome, and in many scenarios it is difficult to select a cleanly differentiated region of the screen to watch, such as when the text is spread around the desktop. Our results show that Capture provides much more complete recording of onscreen text than screen readers such as Orca that are commonly used on commodity desktops.

Screencasting can be used to record the desktop’s display. However, all screencasting approaches produce the same essential result after converting to a standard image or video format. Unlike these approaches, Capture records the structure and semantics of the display-centric text content, including extracting GUI and contextual information from it. This information can then be processed in meaningful ways, such as for supporting visually impaired users, that cannot be done by just taking pictures of the screen without preserving any textual semantics as is effectively done by screencasting. More recently, OpenMemex [120] is a proposed system that extends VNC-based screencasting by using offline OCR to extract text from the recorded data. However, OCR algorithms are too slow to be performed online, and therefore do not facilitate real-time applications including assistive technologies. Additionally, OCR is inaccurate with common screen fonts and when acting on compressed images. OCR
is not event-driven and so must either operate periodically, potentially missing important frames, or waste resources to operate on every frame with any screen changes, even if the text is exactly the same, such as if the mouse cursor has moved.

Various lecture recording systems have recorded lectures on desktop computers. Ziewer [189] recorded screen framebuffers and used offline post-processing to detect and annotate certain features of the presentations, including slide text and slide transitions, to associate text with images. However, the approach relies on OCR to identify text features, which is imprecise and also required more storage space for storing higher-quality screenshots containing text. Stolzenberg and Pforte [161] proposed a hybrid system of symbolic text and metadata recording and screen image recording. It relied on key events recorded by customized presentation software, namely slide changes and input by the lecturer. Unlike Capture, none of these lecture recording systems record onscreen text content from unmodified applications and non-lecture applications.

Gyllstrom [75] also attempts to record a user’s desktop usage history like DejaView, but takes a similar approach to screen readers in how it uses the accessibility frameworks provided by modern operating systems. It shares the same limitation screen readers of only observing the user’s single-point interaction with the system, rather than monitoring changes across the entire screen. Because it does not record all onscreen text, Gyllstrom generates text traces based on the onscreen information and links the traces to static document files to access the remainder of documents that were displayed on the screen. In Capture, there are no implied disk-based files, and all data comes from the screen.
11.3 Operating System Virtualization

Given its various benefits, a number of operating system virtualization systems have been implemented over the past decade. Capsules [148] was an early effort in Sun’s Solaris operating system. This research inspired Zap [124], the earliest system to provide complete, host-independent virtualization in a kernel module without base kernel changes. Our work builds on previous work on Zap, but discusses for the first time various key implementation issues that arise in practice in efficiently supporting complete, host-independent operating system virtualization while preserving scalable operating system performance. Some virtualization systems that do not provide complete, host-independent virtualization have been implemented in Windows [42, 181] without kernel changes, in part by interposing at the DLL level with the associated disadvantages of user-level interposition described in §5.2. Other recent virtualization systems have been implemented by making extensive kernel changes to restructure the underlying operating system [122, 138, 173].

Building on ideas from Capsules, Zap, and other early virtualization systems, some commodity operating systems are gradually incorporating virtualization support by making pervasive changes to the operating system kernel. The implementation of per-process namespaces in the Linux kernel [19, 106] began in 2005 and is still in progress. This effort has affected nearly every corner of the Linux kernel. In contrast, our approach is minimally invasive since it treats the operating system kernel as an unmodified black box. This makes it easier to backport to fit commodity operating system kernels, and the use of a kernel module simplifies deployment. While the precise performance impact of implementing virtualization in the operating system kernel is impossible to measure [19], the demonstrated low overhead of our approach on real applications provides comparable performance.
Interposition has been the subject of extensive research, and is repeatedly relied on by a host of approaches for security and other general operating system extensions, spanning all three approaches: user-level methods [3, 13, 68, 81, 84, 92] and [174] (the latter presenting an excellent analysis of ptrace), kernel modifications [18, 39, 80, 139, 173], loadable kernel modules [59, 61, 65] and hybrid methods [62]. Some of these have pointed out severe limitation of the security of user-level based interposition mainly due to vulnerability to races as well as bypassability. Our work reiterates these deficiencies in the context of virtualization. We have not found any work that deals with the issue of referencing unexported kernel functionality or data structures.

Three of these studies offer valuable insight and guidelines regarding the design of interposition facilities. SLIC [65] raises the need to access kernel state data not explicitly exported (e.g., the PID of the current process). This can help reduce the non-portable component of the code by providing a cleaner interface. Janus [174] comments about interposing on internal state, namely being able to extend some internal data structures, in accordance with our analysis of placing shadow data. Garfinkel [61] focuses on process isolation in the context of security whereas our work pertains to operating system virtualization, which not only isolates but also decouples the process from the operating system instance. Virtualization addresses a superset of the race conditions associated with isolation, introduces new classes of races such as initialization and deletion races, and is required to keep references on operating system resources. Our approach is also concerned with performance considerations in addressing these races, and in particular scalability.

Virtualization enables many more benefits than interposition alone, so its implementation must deal with a broader variety of practical issues. For instance, interposition rarely needs to retain state. Thus, as in the discussion on “Incorrectly Mirroring Operating System State,” Garfinkel suggests avoiding the retention of state in the
monitor. In the virtualization module, we must always be aware of state, either in terms of the virtual state maintained by our module or in the context of the operating system state which might contain references to objects that we track. We are unaware of any work that discusses the implementation details of virtualization in practice.

11.4 Application Checkpoint-Restart

Many application checkpoint-restart mechanisms have been proposed [128, 143, 145]. Application-level mechanisms are directly incorporated into the applications, often with the help of languages, libraries, and preprocessors [17, 175]. These approaches are generally the most efficient, but they are non-transparent, place a major burden on the application developer, may require the use of nonstandard programming languages [86], and cannot be used for unmodified or binary-only applications whose source code is not available.

Library checkpoint-restart mechanisms [129, 164] reduce the burden on the application developer by only requiring that applications be compiled or relinked against special libraries. However, such approaches do not capture important parts of the system state, such as interprocess communication and process dependencies through the operating system. As a result, these approaches are limited to being used with applications that severely restrict their use of operating system services.

Operating system checkpoint-restart mechanisms utilize kernel-level support to provide greater application transparency. They do not require changes to the application source code nor relinking of the application object code, but they do typically require new operating systems [89, 99] or invasive kernel modifications [121, 142, 148, 158] in commodity operating systems. None of these approaches checkpoints multiple processes consistently on unmodified commodity operating systems.
Chapter 11. Related Work

Some recent approaches such as BLCR [48], and CRAK [188] and its successor Zap [124] have been designed as loadable kernel modules that can be used with unmodified commodity operating systems, but these systems have relied on incomplete process state information, do not fully support multiprocess and multi-threaded applications, and require significant application downtimes during checkpoints. Our work builds on previous virtualization architecture [16, 124] and copy-on-write ideas [99] to uniquely provide fast and transparent checkpoint-restart of multiple processes on commodity operating systems.

Operating system virtualization approaches like Capsules [148], Zap [95, 124] and OpenVZ [121] decouple processes from the underlying operating system 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 multi-threaded and multiprocess cooperating desktop applications, moving potentially expensive file system, quiescing, and write-back 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 [66, 85, 90, 130] rely on the hardware’s page protection mechanism exported by the operating system to determine when a page has been written. DejaView uses the same approach but handles operating system interactions not covered by other approaches, and is therefore more complete. Previous process migration mechanisms introduced pre-copying [165] and lazy-copying [187] for reduc-
ing 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 operating system resources and quiescing operating system resources for checkpoint consistency.

Hardware virtualization approaches such as Xen [47] and VMware [169] use virtual machine monitors (VMMs) [131] that can enable an entire operating system environment, consisting of both the operating system and the applications, to be checkpointed and restarted. VMMs can support transparent checkpoint-restart of both Linux and Windows operating systems. However, because they operate on entire operating system instances, they cannot provide finer-granularity checkpoint-restart of individual applications decoupled from operating system instances, resulting in higher checkpoint-restart overheads and differences in how these mechanisms can be applied. For example, unlike DejaView’s checkpoint mechanism, VMMs cannot be used to checkpoint and migrate applications from an operating system that needs to be brought down for maintenance to a machine that can remain up and running to minimize the impact of operating system upgrades on the availability of application services [135].

VMs have been used to log or snapshot entire operating system instances and their applications to roll back execution to some earlier time [22, 50, 88, 171, 179]. 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 [168]. 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.

### 11.5 Distributed Checkpoint-Restart

Library-level distributed checkpoint-restart mechanisms resort to substituting standard message-passing middleware, such as PVM [63] and MPI [154], with specialized checkpoint-aware middleware versions. A good survey on these is found in [145]. To perform a checkpoint they flush all the data communication channels to prevent loss of in-flight messages. Upon restart they reconstruct the network connectivity among the processes, and remap location information according to how the network addresses have changed. Examples include MPVM (MIST) [28], CoCheck [140], LAM-MPI [146], FT-MPI [54], C3 [23], Starfish [7], PM2 [163], and CLIP [30]. These library-level approaches require that applications be well-behaved. They cannot use common operating system services because system identifiers such as process identifiers cannot be preserved after a restart. As a result, these approaches can be used only for a narrow range of applications.

While operating system checkpoint-restart mechanisms provide greater application transparency and typically allow a checkpoint at any time, most of these systems provide checkpoint-restart only for applications running on a single node. ZapC builds on DejaView’s checkpoint-restart mechanism to provide general coordinated checkpoint-restart of distributed network applications running on commodity multi-processor clusters.
Unlike other operating system checkpoint-restart mechanisms, BLCR [48] and Cruz [82] provide support for coordinated checkpoint-restart of distributed applications. However, BLCR does not support checkpoint-restart of socket state and relies on modifying applications to cooperate with checkpoint-aware message passing middleware [48]. BLCR also cannot restart successfully if a resource identifier is required for the restart, such as a process identifier, is already in use.

Cruz [82] also builds on Zap, but is based on an outdated and incomplete Linux 2.4 implementation. It restricts applications to being moved within the same subnet and requires unique, constant network addresses for each application endpoint. It cannot support different network transport protocols but instead uses low-level details of the Linux TCP implementation to attempt to save and restore network state to provide checkpoint-restart of distributed network applications in a transparent manner. This is done in part by peeking at the data in the receive queue. This technique is incomplete and will fail to capture all of the data in the network queues with TCP, including crucial out-of-band, urgent, and backlog queue data. No quantitative results have been reported to show that Cruz can restart distributed applications correctly.

MOVE [162] is a mobile communication architecture that allows transparent migration of live network connections. It virtualizes connections perceived by transport protocols to decouple connections from hosts so that they can be remapped into physical connections using network address translation. While MOVE must be installed on both communication end-points, neither ZapC nor NetCont rely on such non-standard infrastructure. ZapC checkpoints all communicating peers together and can reconstruct all saved connections during restart independently of the original network. NetCont migrates networked applications within a single subnet along with their IP address, avoiding the need to virtualize network addresses altogether. NetCont could leverage MOVE’s mechanism to enable migrations across distinct subnets.
11.6 Desktop Power Management

In Chapter 9 we presented a system to reduce the energy consumption of desktop PCs by putting them in standby sleep state, while allowing their networked applications to maintain continuous network presence even when the computer sleeps. We now discuss other related work in this area.

Desktop PCs and laptops in active state can save energy via reduced power draw at the hardware level [58, 79] and extended idle times at the operating system and application levels [151]. Despite these efforts, typical desktops consume 80-110 W when active and 60-80 W when idle [5, 73]. The sleep states in the ACPI standard [4] allow significantly larger energy saving while allowing quick resume. The typical power consumption reduces to 2-3 W in S3 state (suspend-to-RAM), which is often preferred to the S4 state (suspend-to-disk) due to the much faster resume time.

Many approaches attempt to exploit computer idle periods to put the computer into the power saving S3 sleep state to reduce the power consumption by 95% and up. For these, the biggest challenge is to maintain network presence and connectivity even when the computer is asleep. In the lack of network presence, ongoing connections may be broken and the computer becomes unreachable to new incoming connections. Previous work suggests that users do not voluntarily put their machines in order to avoid such disruptions [5].

Wake-on-LAN (WoL) [101] and Wake-on-WLAN (WoWLAN) [149] use NICs that monitor incoming packets and wake up the host when they detect an incoming “magic” packet. Despite their availability, they have not been widely used due to three technical barriers. First, the remote peer needs to know that the host is sleeping. Second, the peer needs to learn the MAC address of the host, and be able to send it a packet through firewalls and NAT setups. Third, WoWLAN does not work when
a laptop changes its subnet due to mobility. In contrast, NetCont is transparent to remote hosts: since the NCS takes over the IP of the sleeping computer, there is no need for additional network setup. Mobility is not an issue because the networked applications themselves remain stationary.

The common approach to support continuous network presence for sleeping hosts is to use a network proxy to maintain network presence and connectivity in lieu of the host [5, 6, 31, 32, 73, 116, 141, 144, 156]. The network connection proxy monitors incoming packets for the host, and uses WoL to wake up the host when it needs to handle a packet. Each received packet results in one of four actions: directly respond to it, wake-up the host, discard the packet, or queue the packet for later processing when the host is awake. The proxy can be an independent system on the same subnet [73], in the switch or the router [31], or in the host’s NIC [144]. In comparison, NetCont builds on standard tools, provides a superset of proxy functionality, and does not need to wake up the host for packets destined to active networked applications that run on the NCS.

Selective Connectivity [8] proposes to allow a host to choose the degree to which it maintains a network presence, rather than just “connected” or “disconnected”. It relies on assistants to help a host while it sleeps. Assistants perform simple tasks that the host normally handles, such as respond to keep-alives. NetCont avoids the need for such stubs as migration is extremely quick, preventing any significant downtime for the networked applications.

In Somniloquy [5], a secondary low-power processor covers for the main processor of the PC. It goes beyond a regular proxy to support applications such as Bit-Torrent and IM via applications stubs. It depends on stubs and host callbacks that need to be written per application and require intimate knowledge of the application and the protocol. While NetCont’s NCS can be viewed as a secondary processor to the
sleeping host, NetCONT doesn’t require any specialized hardware or modifications to existing applications. In addition, by enabling multiple hosts to aggregate their networked applications on a single external processor, one minimizes the amount of additional hardware needed.

SleepServer [6] was developed based on the experience gained with Somniloquy and provides many of the same benefits, while enabling the use of commodity hardware. Like NetCONT, it relies on a set of centralized servers, which it calls SleepServers, that one’s applications can migrate to. However, like Somniloquy and unlike NetCONT, SleepServer requires small stub applications to be written that handle the network activity of applications to allow only those stubs to be migrated and not the entire operating system. However, this imposes a much larger software engineering burden on developers as they have to redesign their applications into multiple components. In contrast, NetCONT works with unmodified applications and therefore can be used today with minimal deployment effort.

Delay-Tolerant Network (DTN) [55] is an alternative approach to handling disconnection of links and hosts, e.g., as a result of power management. DTN proposes a network architecture and application interface that uses optionally-reliable asynchronous message forwarding that operates as an overlay above the transport layers of the networks it interconnects. However, DTN’s services come at the cost of major reworking both of network infrastructure and of how applications use the network.

More recently, virtual machines were proposed for enterprise environment to migrate an entire idle desktop from the user’s host to a dedicated consolidation server [20, 41]. Unlike NetCONT, these approaches require running the entire desktop within a virtual machine and require moving the entire desktop to the consolidation server, which takes a long time because of the substantial amount of state that needs to be moved when using virtual machines. Furthermore, these approaches are forced to
employ expensive workarounds because today’s hypervisors do not provide ACPI support. Specifically, to enable the desktop to sleep after the virtual machine is migrated to the consolidation server, the desktop must be rebooted with the hypervisor disabled before it can sleep, imposing even longer delays and wasting additional power. In contrast, NetCONT does not migrate entire desktops, only network applications that require continued execution due to network activity, significantly reducing migration times. NetCONT also leverages native ACPI support in existing commodity operating systems, avoiding the need for a reboot. The result is substantially faster sleep times and resulting power savings.
It is not uncommon for desktop users to realize the importance of something they saw earlier as part of their computer use, yet be unable to find it or gain access to it. This dissertation presented DejaView, a new personal virtual computer recorder architecture 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. In building DejaView, we demonstrated a new approach for keeping track of the massive amount of data seen through the desktop, and being able to access this data later even when not explicitly saved.

DejaView provides a unique blend of functionality for display recording, playback, browsing, search, and reviving live desktop execution from any point in time. To provide its unique functionality, DejaView combines three key components: a display-centric recording architecture, a display-centric text recorder, and a live execution recording architecture. All of these components rely on different forms of virtualization to enable powerful new functionality in a transparent manner and without sacrificing performance. DejaView abstracts the computer’s display, onscreen contents, and live execution state by virtualizing the interfaces through which applica-
tions interact with their environment and the operating system. Leveraging existing window system, accessibility framework, and operating system functionality ensures transparent operation without any modifications to any component in the complex software stack of modern commodity systems.

DejaView’s display-centric recording architecture uses display virtualization to decouple display state from the underlying hardware and enables the display output to be redirected anywhere, making it easy to manipulate and record. We show that this provides new functionality to record the viewed output as it was displayed and with the same personal context and display layout, transparently, with high display fidelity, and with minimal overhead. The record can be used to enable PVR functionality of the display, including playback, rewind, and fast forward.

DejaView’s display-centric content recorder, Capture, introduces a form of virtualization that interposes on the accessibility framework available on modern operating systems to cache onscreen text and the respective contextual information. This caching allows efficient access to the entire onscreen contents, mitigating the high performance costs of the accessibility framework while retaining its full functionality. We show that this provides new functionality to facilitate real-time access to onscreen contents from both foreground and background windows, transparently, with high accuracy, and with low overhead. The contents can be recorded and used as an index to the display record based on the displayed text and contextual information captured in the same context as the recorded display. The recorded data can also benefit a variety of problem domains beyond desktop search, including assistive technologies, auditing, and predictive graphical user interfaces.

DejaView’s live execution recording architecture uses lightweight operating system virtualization to create a virtual execution environment that decouples the user’s desktop computing environment from the underlying host. Encapsulating the entire
user’s session in a private virtual namespace is crucial for enabling a live desktop session to be continuously checkpointed, and later revived from any checkpoint. This namespace also hosts the virtual display server to ensure that all display state is correctly saved and that multiple revived sessions can co-exist with their independent display states. We show that this provides a new functionality to continuously record live execution state, transparently, in a globally consistent manner, and with minimal overhead and modest storage requirements. The record can be used to revive past sessions, allowing users to manipulate and interact with the recorded execution state. Checkpointing at a fine granularity, shifting expensive I/O operations out of the critical path, and using various optimizations such as fast incremental and copy-on-write techniques minimize any impact on interactive desktop performance. Combining log-structured and unioning file systems allows fast file system snapshots consistent with checkpoints, enabling revived session with simultaneous read-write usage.

We implemented a DejaView prototype that integrates the three components together into a personal virtual computer recorded. Our implementation illustrates the architecture’s transparency in running with unmodified software stack that includes Linux kernel, Debian Linux distribution, X Windows System, and the GNOME desktop environment. Through an experimental evaluation on a wide range of real-world desktop applications and with real desktop usage, we show the architecture’s ability to provide WYSIWYS functionality fast enough for interactive use and without user noticeable performance degradation.

Going beyond DejaView, this dissertation has also shown how the system’s individual components provide useful building blocks for the broader context of application mobility.

We presented ZapC, a system for transparent coordinated checkpoint-restart of distributed applications on commodity clusters. ZapC can checkpoint an entire dis-
tributed application across all nodes in a coordinated manner such that it can be restarted on a different set of cluster nodes. Checkpoint-restart operations execute in parallel across different nodes to minimize synchronization overhead. Network state is saved in a transport protocol independent manner, for both reliable and unreliable protocols, using standard operating system interfaces. Experimental results demonstrate fast, sub-second checkpoint and restart times of real distributed applications, and modest network state validating the scalability of our approach.

We then presented NetCont, a system that enables a host and its applications to maintain continuous network presence even when the host sleeps, allowing immense energy saving by putting idle hosts in low power sleep state without compromising their connectivity. Using checkpoint-restart, NetCont transparently migrates networked applications off of sleeping hosts to a dedicated server where they continue to run unmodified, relying on the applications themselves to correctly maintain network presence. Applications from hundreds of idle hosts can be consolidated into a single server, reducing energy consumption by orders of magnitude. Experimental results demonstrate fast, sub-second migration times, much lower than host suspend and resume times, and minimal impact on the availability of hosts’ network services.

We also presented Linux-CR, an implementation of application checkpoint-restart that aims for the Linux mainline kernel, and discussed the usage model, the user interfaces, and some key kernel interfaces for the benefit of the entire operating systems community. Building on the experience garnered through this dissertation and on recent support for virtualization available in mainline Linux, Linux-CR’s checkpoint-restart is transparent, secure, reliable, efficient, and well integrated with the Linux kernel. In this, Linux-CR provides for the first time a mature checkpoint-restart platform on a mainstream operating system that opens up new possibilities for researchers, developers and users in the broader community.
These systems demonstrate that our checkpoint-restart architectures, enabled by the virtual execution environment, can provide application mobility and other useful new functionality with low overhead while delivering high performance, and can operate transparently with existing operating systems and unmodified applications.

12.1 Future Work

The work developed in this dissertation raises the possibility for a number of improvements and challenging questions to consider for future research directions.

Our most immediate goal is to continue the active development and implementation of Linux-CR. A key ingredient to a successful upstream implementation is the understanding by the kernel community of the usefulness of checkpoint-restart. That we have received help in contributions, code review and advice, as well as media coverage [37, 38, 51], shows that the usefulness of Linux-CR is recognized. We have high hopes that, with the community’s help, the project will succeed in providing checkpoint-restart functionality in the upstream kernel.

Although DejaView’s execution recording can re-generate state from any past checkpoint, it is not possible to revive sessions from points in time between two successive checkpoints. Therefore, one cannot guarantee that the state in between two consecutive checkpoints will be accurately regenerated. This limitation can be addressed by adding transparent, lightweight, deterministic execution record and replay capabilities such as Scribe [98], to enhance DejaView’s PVR functionality with application execution, including rewind and fast forward of execution state.

On today’s desktops, visual data is being generated increasingly faster than users themselves can consume. Even if captured by our eyes, our brain is often unable to keep up with all of the information flow, and the ability of computers to process the
semantics of visual content lags far behind the rate at which output is generated. Providing access to all of the onscreen contents will enhance the computer’s ability to process display-centric data. Capture’s unique combination of performance and functionality enables new ways of using onscreen content beyond desktop search.

One example is assistive technologies, where screen readers can leverage Capture to provide access to display-centric content previously unavailable to visually impaired users. Recorded information could be used as a virtual hot spot, to notify users of changes to specific desktop entities without explicitly identifying physical screen locations to watch. Users can be notified of events such as changes to specific applications, a status bar that reaches 100% or halts part way for a period of time, or appearance of a particular phrase onscreen. This will enable blind or partially sighted users who rely on audio translation to more easily do out-of-order inspection of a document.

Other examples include auditing of an employee’s desktop session to comply with regulations; enhancing security by employing anti-phishing or intrusion detection tools that monitor the display contents and react to particular patterns (as opposed to rely on network traffic); automated evaluation of user interfaces, such as on websites, by inspecting their visual output; predictive GUI that can learn users’ behaviors and speculate on expected next actions based on screen content hints.

Our work on desktop recording introduces a new approach for information collection, storage, and retrieval. While this dissertation touched upon aspects such as indexing and searching the collected data, information retrieval is a very broad topic with many open issues that warrants further research. In particular, our work opens up new directions to explore in further depth questions such as how to efficiently index the data, what search and ranking strategies to employ, and how to present search results to the user in a meaningful manner.
One challenge is determining the indexing granularity, because there is an inherent tradeoff between runtime efficiency and query result quality. Extracting data at a high frequency may raise the recording overhead and inflate the index size, possibly without real gain in results quality, yet doing so infrequently may result in missed data and poor results quality.

DejaView continuously generates text-shots up to multiple times per second in response to display updates, such that series of text-shots reflect the evolving onscreen content. Because onscreen contents tend to evolve gradually and incrementally, many text-shots are duplicates or near-duplicates. For instance, while each character typed when using a word processor may create a new text-shot, the last text-shot in a typing sequence is often a superset of the previous text-shots. In this case, the “superset” text-shot can faithfully represent the preceding series for indexing purposes.

Therefore, another challenge is developing methods to perform duplicate elimination and aggregation of “similar” data, without compromising the usefulness of the data from the user’s perspective. We envision two forms of near-duplicate elimination: offline and online. The offline version will be applied to the static of text-shots and replace a contiguous series of text-shots with a corresponding representative text-shot. This will shrink the index size and allow for more efficient search. The online version will be applied to text-shots returned by search queries and merge similar results into fewer “meta-results”. This will reduce the amount of relevant results and improve the overall user experience. The similarity criterion may depend on the specific query or even the user intention, e.g., differences in text located away from the queried text may or may not be important.

Effective search and ranking strategies are crucial to make DejaView useful to users in practice. More work is needed on 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. Designing a novel ranking strategy for DejaView that goes beyond relying on time of creation and last access used by other approaches presents many novel challenges. The ranking strategy will have to incorporate the display structure reflected in the text-shots by considering the effect of structure and display layout. In particular, text can be weighted according to its role, e.g., to indicate higher relevance of a document title than a menu button, and according to its onscreen properties, e.g., to prefer results from dominant windows or windows in focus. The ranking strategy will also have to account for noise in recorded information, incorporate forms of information aggregation of “similar” data, and define suitable notions of keyword and window “importance” and “proximity” across the desktop display and across time.

Another open problem is how to integrate DejaView’s search with existing desktop search tools that rank and retrieve individual documents. Desktop search can be useful to augment DejaView’s search results or to illuminate blind spots, e.g., find data saved in a file but that has never been viewed onscreen. Similarly, DejaView’s search may also be integrated with Internet search engines. For instance, we anticipate that users will prefer use DejaView to find information from by a past complex web search to retracing their original sequence of queries. A web search on the user’s query, or even on relevant parts of DejaView’s results, may bring new information that may have not been available during the original search. Furthermore, these search paradigms should integrate seamlessly so that users will not have choose between search system.

We envision conducting user studies for several purposes: to reveal usage patterns to better understand how this functionality will be exploited by users over extended periods of time; to explore how the user interface can be enhanced to better fit daily usage needs; to validate DejaView’s ranking strategies and tune its weights; and to
empirically quantify DejaView’s search performance. User studies could be structured in that users perform predefined tasks and then execute queries for information seen during these tasks, or unstructured to observe real usage and measure the overall utility of the proposed system and how it might save time and enhance user productivity.

Finally, recording a user’s computer activity raises valid privacy and security concerns, as this information could be exploited to infringe upon the user’s civil liberties or for criminal purposes. Addressing the privacy and security ramifications of this emerging computing model is another area that calls for further investigation.
Bibliography


the 13th International Conference on Intelligent User Interfaces (IUI), Sanibel Island, FL, February 2009.


[90] Angkul Kongmunvattana, Santipong Tanchatchawal, and Nian-Feng Tzeng. Coherence-based Coordinated Checkpointing for Software Distributed Shared


[97] Oren Laadan, Dan Phung, and Jason Nieh. Transparent Checkpoint/Restart of Distributed Applications on Commodity Clusters. In *Proceedings of the
IEEE International Conference on Cluster Computing (Cluster), Boston, MA, September 2005.


[121] OpenVZ. http://www.openvz.org/.
CS-97-372, Department of Computer Science, University of Tennessee, July 1997.


[142] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies—a Safe Method to Survive Software Failures. In Pro-


[163] Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, and Yutaka Ishikawa. PM2: High Performance Communication Middleware for Heterogeneous Net-


[183] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad N. Zubair. Versatility and Unix Seman-


