Why Does Windows Crash?

Archana Ganapathi
Computer Science Division
Department of Electrical Engineering & Computer Sciences
University of California, Berkeley

Why Does Windows Crash?*

Archana Ganapathi
University of California at Berkeley
archanag@cs.berkeley.edu

May 20th, 2005

Abstract

Reliability is a rapidly growing concern in contemporary Personal Computer (PC) industry, both for computer users as well as product developers. To improve dependability, systems designers and programmers must consider failure and usage data for operating systems as well as applications. In this paper, we analyze crash data from Windows machines. We collected our data from two different sources – the UC Berkeley EECS department and a population of volunteers who contribute to the BOINC project. We study both application crash behavior and operating systems crashes. We found that application crashes are caused by both faulty non-robust dll files as well as impatient users who prematurely terminate non-responding applications, especially web browsers. OS crashes are predominantly caused by poorly-written device driver code. Users as well as product developers will benefit from understanding the crash behaviors and crash-prevention techniques we have revealed in this paper.

* This work was supported in part by the National Science Foundation, grant CCR-0085899, and the California State MICRO Program. The author was supported in part by a National Science Foundation Graduate Research Fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Table of Contents

Acknowledgements

First and foremost, I would like to thank my advisor, Dave Patterson, for his continual guidance and support. He inspired many ideas during the course of this project. I owe immense gratitude to Brendan Murphy of Microsoft Research for sharing his wealth of knowledge and expertise in failure data analysis. I wish to thank my second reader, Armando Fox, for his advice and constructive criticism on this report. I am grateful to ROC group members and ROC industrial advisors for stimulating discussions and invaluable feedback on this project.

I owe much appreciation to Alex Brown, Mike Howard and Emrys Ingersoll for facilitating crash data collection in the Berkeley EECS department. I am also thankful to volunteers who responded to the EECS department usage survey. Many thanks go to Divya Ramachandran, Steve Stanek and Yang Zhang for their contributions to the BOINC crash collection application. I appreciate all volunteers who contribute their crash data to the BOINC crash collection project.

Last but not least, I wish to express gratitude to my family and friends for providing perpetual moral support and encouragement.

This work was supported in part by the National Science Foundation, grant CCR-0085899, and the California State MICRO Program. The author was supported in part by a National Science Foundation Graduate Research Fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1. Introduction

Personal Computer (PC) reliability has become a rapidly growing concern both for computer users as well as product developers. Personal computers running the Microsoft Windows operating system are often considered overly complex and difficult to manage. We often hear people exclaim, “the Windows operating system is unreliable”. As modern operating systems serve as a confluence of a variety hardware and software components, it is difficult to pinpoint unreliable components. Multiple versions of dynamically-linked libraries (DLLs) and a vast array of peripherals compound errors caused directly by applications developed for the Windows software environment. This complexity precludes manual inspection of crash events to identify features of Windows applications responsible for failure behavior.

Such unconstrained flexibility allows complex, unanticipated, and unsafe interactions that result in an unstable environment often frustrating the user. To troubleshoot recurring problems, it is beneficial to data-mine, analyze and document every interaction for erroneous behaviors. Such failure data provides insight into how computer systems behave under varied hardware and software configurations. To improve dependability, systems designers and programmers must consider failure and usage data for operating systems as well as applications. Common misconceptions about Windows are rampant. Our study attempts to shed some light on the factors affecting Windows PC reliability based on data collected from hundreds of PCs.

1.1 Motivation

“If a problem has no solution, it may not be a problem, but a fact, not to be solved, but to be coped with over time.”

–Shimon Peres

Most Windows users have experienced at least one “bluescreen” during the lifetime of their machine. With the availability of a wide range of downloadable software, there is no reason to hold back and use PCs in a conservative fashion. As a result, application crashes are a common side-effect. A sophisticated PC user will accept Windows crashes as a fact and attempt to cope with them, rather than remain puzzled. However, a novice user will be terrified by the implications of a crash and will continue to be preoccupied with the thought of causing severe damage to the computer.

From a research perspective, the motivation behind failure data-mining is manifold. First, it reveals the dominant failure cause of popular computer systems. In particular, it identifies products that cause the most user-frustration, thus rekindling our efforts to build stable, resilient systems. Furthermore, it enables product evaluation and development of benchmarks that rank product quality. These benchmarks can influence design prototypes for reliable systems. Most importantly, such methodology helps formulate and address research issues in computer system reliability.

Within the realm of an organization, knowledge of failure data can improve quality of service. Often, corporations collect failure data to evaluate causes of downtime.

In addition, they perform cost-benefit analysis to improve service availability. Some companies extend their analyses to client sites by gathering failure data at deployment locations. For example, Microsoft Corporation collects crash data for their Windows operating system as well as applications used by their customers. Unfortunately, due to legal concerns, corporations such as Microsoft will not share their data with academic research groups. Companies do not wish to reveal their internal vulnerabilities, nor can they share third party products' potential weaknesses. While abundant failure data is generated on a daily basis, very little is readily sharable with the research community.

1.2 Contributions and Roadmap

This report presents an exploration of crash behavior in PCs running Windows XP. We provide a brief description of various types of crashes and details on the crash data we collect in section 2. While much related work exists in the area of systems failure data analysis, as presented in section 3, our primary contribution to this research area is Windows XP crash data and analysis. We collect and compare crash data from two different data sets as described in section 4. We use two different data collection mechanisms – one of which we developed ourselves at Berkeley. These data collection tools are outlined in section 5. We describe tools and techniques we use for our data analysis in section 6.

We analyze the underlying causes for both application-level as well as operating system-level crashes. We also compare application crash data to application usage statistics collected from our users. In our study of application crashes (section 7.1), we have identified web browsers as the single most crashing application type in the Windows environment. We found that application crashes are caused by both faulty non-robust dll files as well as impatient users who prematurely terminate non-responding applications. Operating system-level crashes, discussed in section 7.2, are predominantly caused by poorly-written device driver code. Section 7 explains the above (and more) analysis results and outlines potential techniques to reduce crashes in PCs. In section 8, we discuss shortcomings of our analysis, including missing information that would compliment our data analysis and allow us to improve our understanding of Windows crashes. As a solution for overcoming hurdles we encountered in collecting data from various sources, we propose an open source repository for failure data, details of which are outlined in section 8.2. Section 9 concludes.

2. Background

To study Windows crash behavior, we collect data, in the form of *crash dumps*, from two different sources (discussed in section 4), using two different collection mechanisms (discussed in section 5). We study various types of crashes, which differ in their manifestation as well as impact to the user. Each type of crash is defined and explained in section 2.1. The amount of information collected to analyze each type of crash varies based on the data collection mechanism used. We discuss the contents of crash dumps and other information collected in section 2.2. We parse each of the collected crash dumps using Windows debugging tools (as described in section 6.1) and analyze the data to understand crash patterns in Windows machines.

2.1 Crash Definitions

There are various types of "crashes" that a Windows user may encounter. These crash-types vary in their manifestation and their impact on the user's experience. We define each crash-type below:

- **Crash** – An event caused by a problem in the operating system(OS) or application(app) requiring OS or app restart.
- **Application Crash** – A crash occurring at user-level, caused by one or more components (.exe/.dll files), requiring an application restart.
- **Application Hang** – An application crash caused as a result of the user terminating a process that is potentially deadlocked or running an infinite loop. If the user intervenes to terminate the process, the component (.exe/.dll file routing) causing the loop/deadlock cannot be identified.
- **OS Crash** – A crash occurring at kernel-level, caused by memory corruption, bad drivers or faulty system-level routines. An OS crash includes blue-screen-generating crashes, which require a machine reboot, as well as Windows explorer crashes, which require restarting the explorer process.
- **Bluescreen** – An OS crash that produces a user-visible blue screen followed by a non-optional machine reboot.

2.2 What are crash dumps

Upon each application crash or bluescreen generated by the operating system, Windows collects failure data as a *minidump*. Users have three different options for the amount of information that is collected upon a crash. We use the default (and smallest) option of collecting small dumps, which are only 64K in size. These small minidumps contain a snapshot of the computer's state at the time of crash. They include a list of loaded drivers, the names and timestamps of binaries that were loaded in the computer's memory at the time of crash, the processor context for the stopped process, and process information and kernel context for the stopped process and thread as well as a brief stack trace. We do not collect personal data files for our study. However, portions of such data may be resident

in memory at the time of crash and will consequently appear in our crash dumps. For further details on the contents of crash dumps, the interested reader can refer http://support.microsoft.com/kb/254649/

When an OS crash occurs, typically the entire machine must be rebooted. Any relevant information that can be captured before the reboot is saved in a .dmp file in the %windir%\Minidump directory. These minidumps are uniquely named with the date of the crash and a serial number to eliminate conflicting names for multiple crashes on the same day.

When an application crashes, the user typically receives a prompt asking if they would like to send the crash-related information to Microsoft. The information that is collected includes a minidump as well as a list of all modules loaded by the crashing process. Unlike OS crashes, application minidumps are stored in application-specific locations and are often difficult to locate on a machine. To increase the amount of data we receive, we disable the data-requesting prompt and automatically collect data for every application crash on the user's machine.

3. Related Work

Jim Gray's work [Gra86, Gra90] serves a role model for most contemporary failure analysis work. Gray did not perform root cause analysis but rather *Outage Cause* that considers the last in the fault chain. In 1989, he found that the major source of outages was due to software, contributing to about 55%, far outrunning its immediate successor, system operations that contributed 15%. This observation led him to blame software for almost every failure; it was supposed to mask all single faults. We study software (application) crashes as well as system crashes and understand the cause and effect of both crash types.

Deviating from Gray's outage cause analysis, in our study we perform root cause analysis under the belief that the first crash in a sequence of crashes is responsible for all subsequent crashes within that event chain. The past two decades have produced several studies in root-cause analysis for operating systems (OS) ranging from Guardian OS and Tandem Non-Stop UX OS to VAX/VMS and Windows NT [Gra90, Kal98, LI95, SK+00, SK+02, TI92, TI+95]. In server environments, Tandem computers, VAX clusters as well as several operating systems and file servers have been examined for software defects by several researchers. Lee and Iyer focussed on software faults in the Tandem GUARDIAN operating system [LI95], Tang and Iyer considered two VAX clusters running the VAX/VMS operating system [TI92], and Sullivan and Chillarege examined software defects in MVS, DB2, and IMS [SC91]. Murphy and Gent also focussed on system crashes in VAX systems over an extended period, almost a decade [MG95]. They concluded that system management was responsible for over 50% of failures with software trailing at 20% followed by hardware that is responsible for about 10% of failures. While examining NFS data availability in Network Appliance's NetApp filers, Lancaster and Rowe attributed power failures and software failures as the largest contributors to downtime; operator failure contributions were negligible [LR01]. Thakur and Iyer examined failures in a network of 69 SunOS workstations [TI96]. They divided problem root causes into network, non-disk and disk-related machine problems. Kalyanakrishnam et al. perused six months of event logs from a LAN comprising of Windows NT workstations that delivered emails [KK+99]. Using a state machine model of detailed system failure states to describe failure timelines on a single node, they concluded that most automatic system reboot problems are software-related; the average downtime is two hours. Similarly, Xu et al. considered Windows NT event log entries related to system reboots for a network of workstations that were used for enterprise infrastructure, allowing operators to annotate event logs to indicate the reason for reboot [XK+99]. In this progression, our study of Windows' crash data gauges the evolution of PC reliability. We compare these results with similar information from earlier systems. Koopman et al. test operating systems against the POSIX specification [KD00]. Our study is complimentary to this work as we consider actual crash data that leads to OS unreliability.

Recently, in Windows XP Machines, Murphy deduced that display drivers were a dominant crash cause and memory is the most frequently failing hardware component [Mur04]. We extend this work, evaluating application crashes over and above operating system crashes. We study actual crash instances experienced by users rather than

injecting artificial faults as performed by fuzz testing [FM00]. This study of crash data differs from error log analysis performed by Kalakech et al. [KK+04]; we determine the cause of crashes in addition to time and frequency.

Applications constantly evolve with enhanced features and more protection mechanisms to safeguard from potentially unsafe environments. Despite a proliferation of techniques to improve reliability, PC components continue to fail, causing much user frustration. These systems offer potentially fruitful avenues for research with the promising potential for many practical suggestions to improve the performance of software for system designers and developers. However, some researchers argue that the key property of a well-conditioned system is *graceful degradation* [WC+01]. This trait has not been achieved in most PC applications; crashing is far from graceful degradation. We attempt to understand the reason behind such behavior in Windows applications.

A fast technique to detect and recover from software errors is continuous testing of the software with various inputs. However, devoid of clairvoyance it is usually far from obvious which inputs to throw at complex systems. Many researchers use fault injection to perform post-deployment prophylactic tests. Injected faults include data corruption, such as flipped bits in registers or memory and *stuck-at* faults, code corruption, such as op-code alteration and incorrect call routes [SM+04, BS+02], as well as performance faults. In the absence of real failure data, fault injection is a good alternative. However, we collect actual crash data from numerous users to study and evaluate PC software.

Several researchers have provided significant insights on benchmarking and failure data analysis [BC+02, BS97, OB+02, WM+02]. Wilson et al. suggest evaluating the relationship between failures and service availability [WM+02]. Among other metrics, when evaluating dependability, system stability is a key concern. Ganapathi et al. examine Windows XP registry problems and their effect on system stability [GW+04]. Levendel suggests using the catastrophic nature of failures to evaluate system stability [Lev89]. Brown et al. provide a practical perspective on system dependability by incorporating users' experience in benchmarks [BC+02, BS97]. In our study of crashes, we consider these factors when evaluating various applications.

4. Description of Data Sets

A single data set can be construed as an imprecise representative of typical Windows computer usage. For example, academic/corporate computer users have a level of computer expertise higher than average computer users. To reduce the skew introduced by a single data source, and to increase variability in usage profile, we consider several different data sources and attempt to understand the biases/assumptions implicit in each data set. We describe these data sets below.

4.1 UC Berkeley EECS Department

Our primary data collection and analysis was performed on research machines in the EECS department at UC Berkeley. Since June 2004, over 200 machines that run Windows XP SP1 are reporting their crashes to our server. The users of these machines are professors, graduate students and/or departmental staff/admins.

These machines operate within the same domain and are somewhat constrained in security and administration. Much of the software installed in these machines is available internally to all EECS users. Thus, it is safe to say that these applications are somewhat stable and widely used in the department. However, users have the ability to install any software they require, and many of the graduate students use custom-written software that their research group has produced. The system administrators do not restrict use of such software, but ensure that necessary safety precautions are taken and patches are updated.

We have collected data since mid-June 2004 and will present analysis for 10 months of data (see Figure 1). We incrementally added computers to report to us (to verify stability of the collection mechanism). Also, since we are in an academic setting, we must account for gaps in crash data due to holidays and semester breaks. Below, we provide a timeline of such events/milestones during our 10 months of data collection:

Jun 14: 25 machines
Jun 25: 125 machines
July 9: 150 machines
Aug 3: 214 machines
Aug 24: Fall semester begins
Nov 25-26: Thanksgiving break
Dec 21-Jan 10: Winter break
Jan 11: Spring semester begins
Mar 21-Mar 25: Spring break

Given that the data is collected from a population of experienced (and perhaps expert) computer users, we realize the crash data we receive from this group might be biased and may not accurately represent the PC user population as a whole. We attempt to address this issue using BOINC, as described in section 4.2.

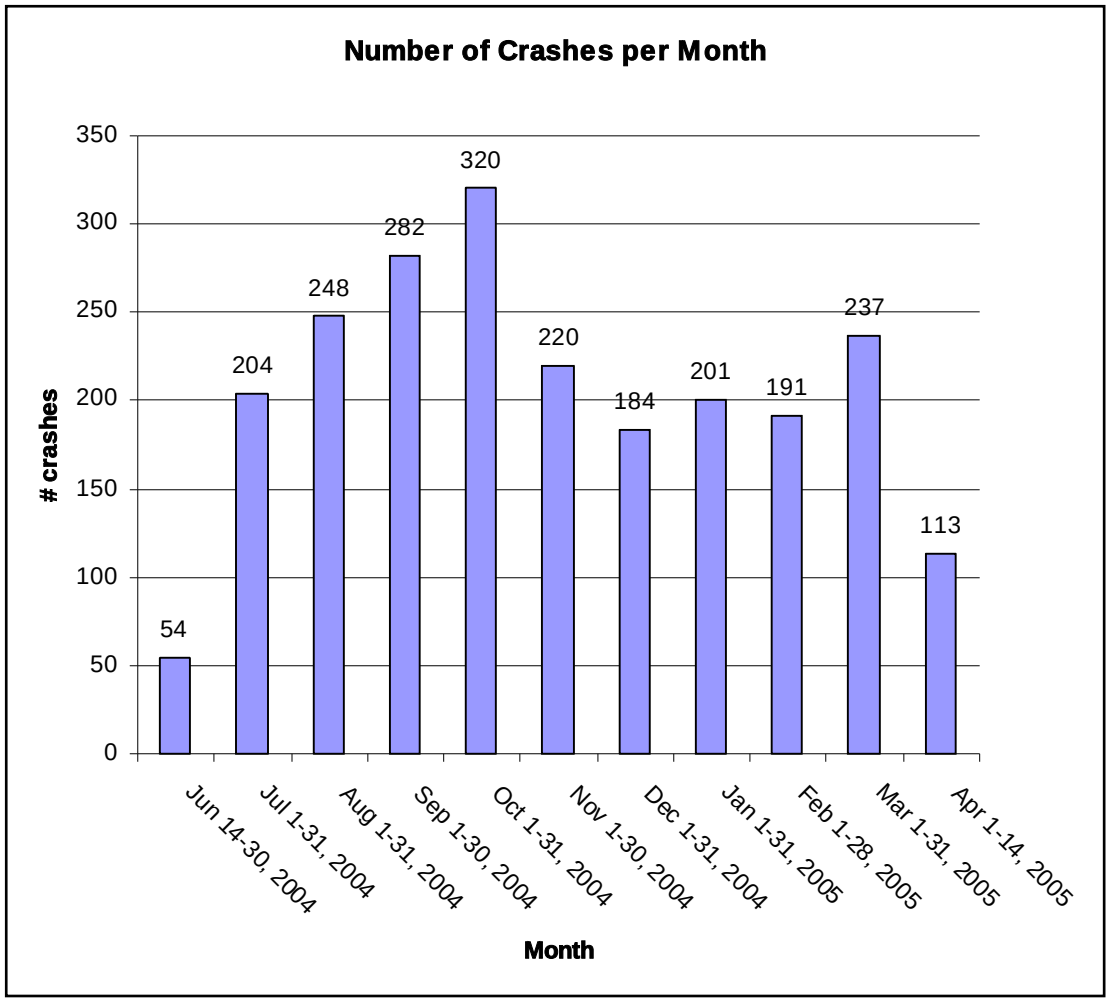

Figure 1: Number of crashes reported per month. This graph is based on the data collected in the UC Berkeley EECS department.

4.2 BOINC User Group

To study a broader population of Windows users (in a less constrained environment), we have embarked on an effort to target public-resource computing volunteers. BOINC is a platform for pooling computer resources from volunteers to collect data and run distributed computations [And03]. A popular example of an application using this platform is SETI@home, which aggregates computing power to 'search for extraterrestrial intelligence'. Numerous people enthusiastically contribute data to projects on BOINC rather than corporations as they favor a research cause. Additionally, users appreciate incentive either through statistics that compares their machine to an average BOINC user's machine, or through recognition as pioneering contributors to the project.

Currently, we have about 150 BOINC users. We are working on publicizing this effort further. So far we have received 562 OS crashes from these users, which we analyze to understand the types of and implications of OS level crashes.

5. Data Collection process

We use two different mechanisms to collect crash data. To collect data from machines within the same administrative domain, we use Microsoft's Corporate Error Reporting tool. Data collection for machines that reside in different domains is done using BOINC, as described in section 5.2.

5.1 Corporate Error Reporting (CER)

To collect data, we use Microsoft's Corporate Error Reporting (CER) software. We configure a server with a shared directory that can directly receive crash reports from other machines within the same domain. Reporting client machines require no additional software. We simply modify a few registry entries (using a group policy) to redirect crash reports to our server in place of Microsoft. Furthermore, we disable the prompt that asks users whether they wish to send a crash report. Thus, we are guaranteed to receive reports for all crashes and are not dependent on the good graces of the user to send us crash data. Figure 2 shows sample information logged for each crash reported.

9/2/04 21:58	M1	Usr1	iexplore.exe\6.0.2800.1106\rpcl3260.dll\6.0.9.1575
9/2/04 21:59	M2	Usr2	notepad.exe\5.2.3790.0\comctl32.dll\6.0.3790.0
9/3/04 0:31	M3	Usr3	sgtray.exe\1.0.89.0\anigifdisplay.ocx\1.0.89.0
9/3/04 0:46	M4	Usr4	excel.exe\9.0.0.3822\excel.exe\9.0.0.3822
9/3/04 0:57	M5	Usr5	NOTEPAD.EXE\5.1.2600.0\hungapp\0.0.0.0
9/3/04 1:19	M6	Usr6	iexplore.exe\6.0.2800.1106\unknown\0.0.0.0
9/3/04 1:30	M5	Usr5	win-ir pro.exe\3.4.25.1\win-ir pro.exe\3.4.25.1
9/4/04 0:19	M7	Usr7	CDCopier.exe\5.3.5.10\hungapp\0.0.0.0

Figure 2: Sample data extracted from CER crash reports. The first column shows the time of crash. The second and third columns represent the anonymized machine and user name. The last column shows the crashing application, application version, crash-causing component, and component version.

The CER server collects all the crash reports from each machine and sorts them by application. Each crash report is a .cab file containing the crashdump, a text description of the crashing application and it's version, and a log of the number of times the same time of crash occurred on that machine.

The convenience of using CER is that it provides user-friendly interfaces for configuration as well as viewing/organizing crash reports. A major drawback of this approach is that all reporting machines must be in the same network domain as the CER server. This structure limits us to aggregating data from a single organization at a time. Furthermore, we cannot collect any historical data from client machines (such as crashes that occurred prior to CER installation). Additionally, there is no usage information collected by CER. Any usage-related metrics must be collected using an orthogonal mechanism, which often makes it inconvenient to correlate with crash data.

5.2 Berkeley Open Infrastructure for Network Computing (BOINC)

Berkeley Open Infrastructure for Network Computing (BOINC) provides services to send and receive data from its users via the HTTP protocol using XML formatted files. It allows application writers to run and maintain a server that can communicate with numerous client machines through a specified Applications-Programmer-Interface (API). Each subscribed user's machine, when idle, is used to run BOINC applications. Project groups can create project web sites with registration services for users to subscribe and facilitate a project. The web site can also display statistics for contributing users.

Taking advantage of these efforts, we have created a data collection application to run on this platform. BOINC provides a good opportunity to collect and aggregate data from users outside our department while addressing privacy concerns. We have written tools to read crash dumps from users' machines and send the data to our BOINC server. In addition, we are also able to collect usage data with users' consent. The drawback of this mechanism is that we can only collect crash dumps that are stored in known locations on the user's computer, consequently excluding application crash dumps that are stored in unknown app-specific locations. Furthermore, configuring the BOINC server is a tedious and meticulous task. We must also monitor the number of *work units* we allot for the BOINC projects; if there are not enough *work units*, the application will not run on client machines.

An attractive aspect of using BOINC is that we can add more features to our application as and when necessary. We can also provide users with personalized feedback pages, consequently rewarding the users with an incentive for sharing data. However, we must verify the integrity of each crashdump we receive from the users. We must safeguard our server from being sabotaged by malicious data responses.

6. Crash data analysis

We use a combination of Microsoft's analysis tools and custom-written scripts to parse, filter and analyze the crash data. We provide an overview of these tools in the next few sections.

6.1 Description of analysis tools

Upon receipt of crash dumps, they are parsed using Microsoft's "Debugging Tools for Windows" (WinDbg), publicly available at http://www.microsoft.com/whdc/devtools/debugging/default.mspx. We retrieve debugging symbols from Microsoft's publicly available symbol server (http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx). Parsing crash dumps using WinDbg reveals the application in which the crash was experienced as well as the immediate cause of the crash via an error code of the crashing routine. The drawback of this approach is that we rely on the completeness and accuracy of Microsoft's symbols. Due to legal reasons, Microsoft does not make 3rd party symbols available so we cannot rely on our current tools to provide an accurate stack trace for 3rd party applications; the issue is that we may not accurately identify the component causing the application crash even thought the application that crashed is identified correctly.

Once crash dumps are run through WinDbg, the importance of filtering data is evident. When a computer crashes, the application and/or entire machine is rendered unstable for sometime during which a subsequent crash is likely to occur. Specifically, if a particular component of an application, such as a dynamic-link-library (.dll) file is corrupt, the application is likely to repeatedly reproduce the error. It is inaccurate to double-count subsequent crashes that occur within the same instability window. To avoid clustering unrelated events while capturing all related crash events, we study the number individual crash events forced into clusters using various temporal windows.

The data that is collected can be used to gather a variety of statistics. We can provide insight to the IT team about the dominant cause of crashes in the organization and how to increase product reliability. We can also use crash behavior to track any potential vulnerabilities as frequent crashes may be a result of malware on the machine. In the long run, we may be able to develop a list of safe and unsafe applications (and versions) and which combinations of concurrent installations result in crashes.

6.2 Clustering the data

For the purposes of our study, we identify three states for each application installed on a computer – not running, running (without problems) and crashed. Our data does not allow us to identify the *not running* and *running* states; we only have information about crashes. Based on our own PC usage experience, Figure 3 shows the behavior of concurrently running applications on our PC.

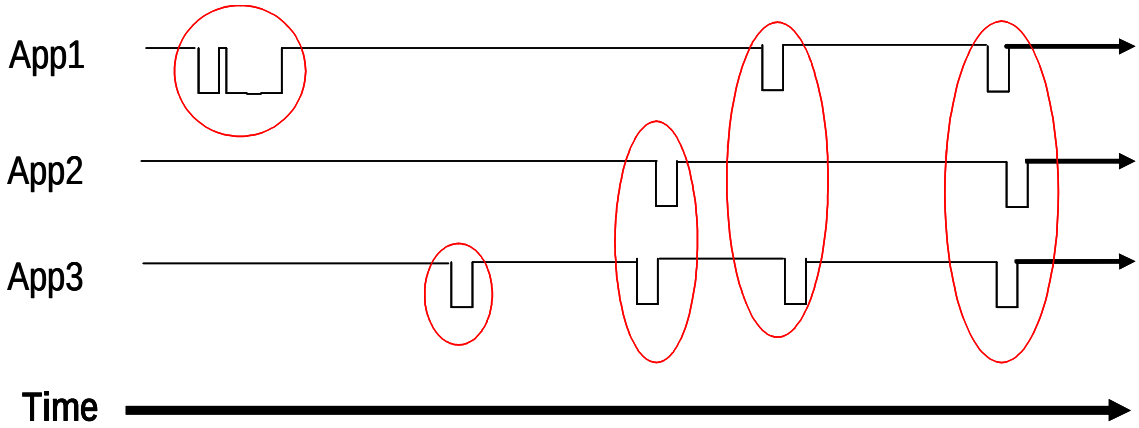

Figure 3: Behavior of concurrently executing applications on a single computer. Typically, a single application crash triggers several subsequent crashes. These events are grouped by inter-crash times.

We often have different applications open in different *windows* in modern PCs. Even if only one application is actively being used, the other applications continue to run in the background until they are explicitly terminated. Often, our interactions and the consequent behavior of one application affects the behavior of other concurrently running applications or subsequent instances of the same application. There may be several reasons for such cascading effect (or clustered crash behavior), a few of which are outlined below:

- **Shared resources -** Applications often share some common resources such as CPU and memory. If one application exhausts the memory available on the PC, other applications are impacted and slow down significantly (perhaps even stop responding altogether).
- **Dependant processes -** In some scenarios, one application may fork a process to invoke another application. For example, when using MS Outlook for e-mail, MS Word is invoked as the default editor for composing messages. If the parent Outlook application crashes, then the child process used for text editing in Word is directly affected.
- **System instability –** When there is a persistent problems, such as hardware failure, software misconfiguration, or underlying operating system instability as a result of a virus attack, all applications running on that machine are impacted. Often, in the absence of anti-virus software, repeated crashing behavior can indicate the existence of a virus on the system.
- **User retry** – When a user-initiated action is unsuccessful, the user often retries the same action until it is successful or they lose patience and try alternative means to accomplish their task.

In all the above scenarios, we must consider each *cluster* of crashes as a single event (initiated by the first crash event). Counting each crash event separately (without grouping them) leads to false accusations and skewed results. Since we do not know the exact sequence of events in every crash cluster scenario, we try to extrapolate based on the time between crash events on each individual machine. Figure 4 shows the number of individual crash events filtered out when clustering crashes based on the various time

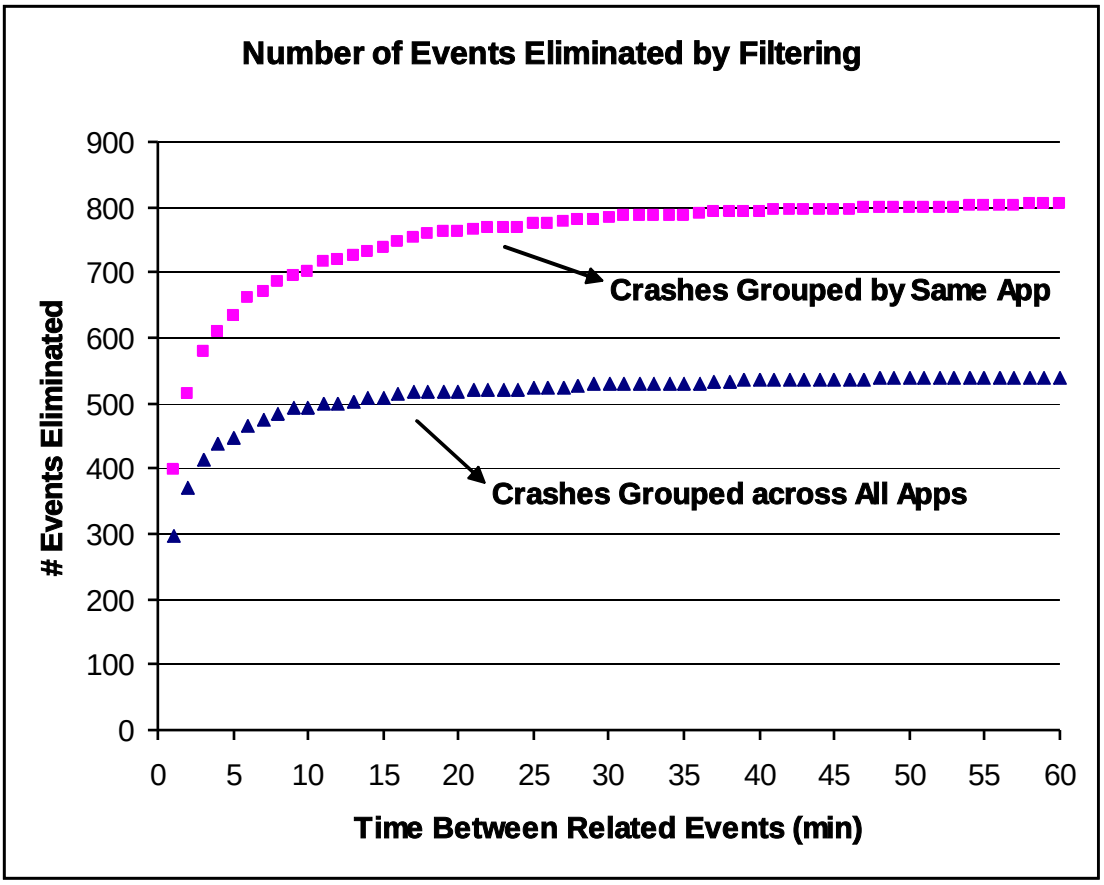

Figure 4: Crash data filtering by time. This graph shows the number of crash events filtered out by setting the time between related events to the value indicated on the x axis.

intervals to determine related events. We try clustering events that occur across different applications on the same machine as well as limiting clusters to single applications.

When choosing the appropriate time interval to determine crash event clusters, there are several concerns to address. Choosing a very long time interval would falsely impose relationships across applications. Given a large enough time interval, all crashing applications would be considered related. On the other hand, choosing a very short interval would perhaps overlook many valid inter-application relationships. Thus, based on trends observed in the graph in Figure 4, we chose a 10 minute time filter. In our data set, crash events that occur on the same machine within 10 minutes of each other are considered related. For subsequent data analysis sections in this paper, we only consider the first crash event in each crash cluster. This clustering technique reduced the 2254 crashes into 1521 crash clusters.

Another side-effect of crash clustering is that we eliminate duplicate records for the same event. Often, if an application is not responding and the user tries to manually terminate it (for example, by clicking the 'x' on the top right corner of the window), they may not see a prompt to terminate the process for a few seconds. Some users impatiently click the 'x' multiple times, consequently receiving several prompts to terminate the same application. This scenario may generate several records for the same crash event. Thus, it is important to soften the effect of such outliers in our data set.

We also experimented with automatically clustering application crash events using statistical learning theoretic algorithms. We augment the crash data with information about usage patterns and program dependencies and feed the data into the *k-means* and *agglomerative clustering* algorithms to determine which applications are behaviorally related. Preliminary results highlight the importance of identifying features of collected crash data that provide information about program structure, system configurations, and user behaviors, and defining distance measures for clustering that use those features effectively. For more details, see Appendix B.

7. Analysis Results

Crashes, at a high level, can be partitioned into two types – application-level and kernel-level/OS crashes. To fully understand the dynamics of Windows crashes, we studied both types of crashes to the extent facilitated by the analysis tools available to us. In the EECS data set, only 79 of the 1521 crashes were caused by the OS (these OS crashes include bluescreen-generating crashes as well as Windows explorer crashes). The remaining 1442 crashes were application-level crashes. Sixty of these 79 OS crashes were caused by Windows Explorer; however, these Explorer crashes occurred at the application level and did not generate blue-screens or kernel-level problems. Thus, despite the fact that Explorer is a Windows OS related component, we analyze Explorer crashes among other application crashes. The remaining 19 OS crashes were due to blue screens generated by various drivers operating with kernel-level capabilities. A more detailed discussion of OS crashes follows in section 7.2, as BOINC has many more OS crashes.

In the BOINC data set, we were able to collect 562 OS crashes from 77 users (we have many more users voluntarily reporting crashes to us through BOINC; however, not all of them have experienced OS crashes). We also collected a handful of application crash dumps. However, all these application crashes were related to Microsoft-written applications such as notepad and MS word (as those were the only crash folders we were able to locate). Consequently we do not analyze these application crashes from the BOINC data set.

7.1 Application Crashes

Modern PCs run a wide variety of application software. There are hundreds of thousands of different applications available to PC users, each with numerous versions supporting a variety of features. It is difficult for the operating system to support such a spectrum of application requirements and workloads. Consequently, it is easier for applications to misbehave or simply behave in a manner that is not anticipated by the user. In our data set from the UC Berkeley EECS department, applications are responsible for over 95% of the crashes. In the remainder of this section, we elaborate on applications and their crash behavior.

7.1.1 How Do We Categorize Applications?

As a result of our automatic clustering experiment, we determined that we did not have enough data to derive a method to categorize applications in our data set. There was no unifying theme for crashes grouped together by our clustering algorithm. (See Appendix B for details) For example, we had:

- crashes from the same user/machine
- crashes from the same application
- crashes from similar applications (based on what they were used for)
- crashes from application written by the same organization

For the sake of simplicity, we chose to impose a categorization based on application functionality i.e. what they are typically used for. We describe each

application category we use and provide example applications that would fall under each category.

- **code development** – Applications in this category, such as Visual Studio and Java Eclipse, are primarily used for the purpose of writing custom software. These applications rely on various libraries for providing a variety of functionality to the user.
- **custom software** – This category entails applications that are developed by the users for themselves/other users. Several research groups in the Berkeley EECS department develop software to assist their work and/or for the benefit of the industry. Thus, it is common to see these tools being used in the EECS department. Unlike commercially available software, custom software is statically linked and does not depend on many dynamic link libraries.
- **database** – This category includes typical database such as SQL Server and MS Access. The primary function of applications in this category is to provide an interface to organize and access data stored in a repository.
- **document and presentation editing** – Applications in this category, such as MS Word, LaTeX and MS Powerpoint, are used as a means create and modify textual documents and/or presentations. These applications are widely used as they often increase efficiency for the user by providing facilities such as spelling checks.
- **document archiving** – This category of applications include gzip and MS cab extractor, whose primary function is to compress documents for efficiently archiving them and uncompress documents to view them. Typically, these applications are used rarely.
- **document viewing** – Applications such as Adobe Acrobat Reader and Ghostview serve the main purpose of document viewing. These applications are not used for editing/updating documents, and thus provide a read-only interface to the user.
- **e-mail** – There are a plethora of applications used for reading/writing e-mail. Common examples include MS Outlook, Eudora and Thunderbird.
- **I/O** – This application category includes all software used for interfacing with I/O devices such as scanners, printers, and handheld device. For instance, users "hotsync" data between their computer and handheld devices such as a palm pilot.
- **instant messaging** – Numerous applications have been developed to enable users to communicate instantly with other peers online. Examples of applications in this category include AOL Instant Messenger, Yahoo Messenger, and MSN Messenger. These applications rely on underlying network libraries to connect to a server or directly connect to peer machines for sending text messages and/or documents.
- **multimedia** – Several applications facilitate recording and/or playback of audio and video files. Media players such as Real Audio Player and Windows Media Player also allow users to stream files from remote locations. Thus, these multimedia applications often depend on network libraries in addition to audio-visual libraries.
- **remote connection** – Many users prefer to work off-site, especially at night and on weekends. Additionally, many users may need to log into machines that are

located in a server room, and the only mechanism to access them is to use applications such as sshclient and exceed, which enable remote connectivity.

- **scientific computation** – Applications in this category include Matlab and Mathematica, which are typically used perform complex arithmetic. Consequently, these applications tend to be very CPU-intensive.
- **system management and security** – Few computer users perform minimal maintenance on their computers. They use software such as Microsoft Management Console to manage devices and other software such as SQL Server. With the increase of malware, tools such as stopzilla assists the user in removing spyware and pop-ups. Applications such as Microsoft Management Console and stopzilla are considered in this category.
- **web browsing** – The most common application used in PCs is web browsing software. Examples of applications in this category include MS Internet Explorer, Netscape and Firefox. Several web browsers come with embedded e-mail clients. However, for the purposes of our study, we do not consider such e-mail clients in the e-mail category as it is extremely difficult to distinguish them from the actual browser components.

Figure 5 suggests expertise levels of typical users who use applications in each of the categories mentioned above. While it is possible to categorize crashes based on the likely expertise of a typical user of the application that crashes, we realized such analysis would not reveal information detailed enough for application developers to react to. Categorizing applications based on how they are used will not only provide insight to users of these applications; it will also reveal shortcomings in the underlying design flaws in inter-component interactions in each application category.

Application Category	**Novice user**	**Intermediate User**	**Expert User**
Code development	No	No	Yes
Custom software	No	No	Yes
Database	No	Yes	Yes
Document presentation and editing	Yes	Yes	Yes
Document archiving	No	Yes	Yes
Document viewing	Yes	Yes	Yes
e-mail	Yes	Yes	Yes
I/O	Yes	Yes	Yes
Instant messaging	Yes	Yes	Yes
Multimedia	Yes	Yes	Yes
Remote connection	No	Yes	Yes
Scientific computation	No	Yes	Yes
System management and security	No	No	Yes
Web browsing	Yes	Yes	Yes

Figure 5: Typical computer expertise level of people who use applications in each category.

7.1.2 How Can a Usage Survey Help Interpret Crash Behavior?

In each application category, a handful of applications caused a majority of crashes in that category. However, it is unfair to judge the quality and/or reliability of applications based solely on crash count. In the absence of a process monitoring application usage, we conducted a survey among users whose machines generated crashes in our data set. The questions we asked our survey-takers are available in Appendix A. We received over 50% of the responses (41 responses). Due to the nature of surveys, we cannot rely on the responses to gain an accurate understanding of how frequently these applications are used. However, we can use survey responses to approximate the correlation between usage and crash behavior and justify crash patterns based on usage trends. We acknowledge that the difficulty of objective evaluation of computer usage taints our survey responses. However, though less accurate than automated monitoring, the information we gathered highlights unusual occurrences in application crashes.

Given that the EECS department contains a wide variety of users, we try to catalogue the affiliations of these users in Figure 6. The computer-expertise of these users spans a fairly wide range (from experts who build their own machines to application-level users). The quantity (and types) of crashes generated by each type of user varies based on their usage level. Graduate students, for example, often develop their own software and may cause several crashes in the process of debugging. We would consider such users as experts. The department's administrative staff, on the other hand, typically limit their use to pre-existing applications and do not experiment much with their machine. They may range from novice to intermediate users.

Type of User	Number of Users	Number of Crashes
graduate student	30	621
staff	28	414
unknown	16	197
faculty	14	191
undergraduate	4	19
visitor	3	51
guest	1	9
postdoc	1	19
TOTAL	**97**	***1521***

Figure 6: User Profile in the UC Berkeley EECS data set.

Figure 7 and Figure 8 show the usage proportion and number of crashes, respectively, on each day of the week. All the EECS department users who responded to our survey use their EECS computers Monday through Friday. Very few of them use these computers on weekends. Crashes, on the other hand, do not occur uniformly across the five days of the working week. There appear to be far fewer crashes on Fridays than Monday through Thursday. This trend may be due to the fact that Fridays tend to be more relaxed than the other four work days as many people simply wrap up work from the previous days. Saturday and Sunday naturally have very few crashes as many people do not come to the department to work on those weekends.

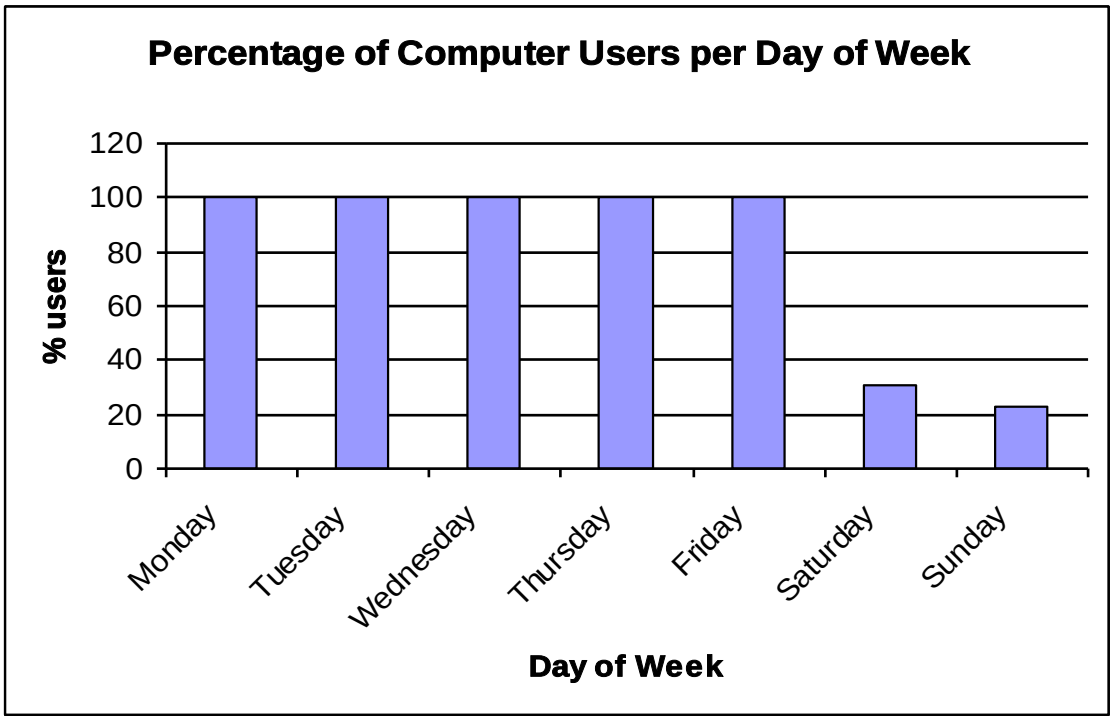

Figure 7: Computer Usage by Day of Week. This graph depicts the percentage of our survey responders that use their EECS computer on each day of the week.

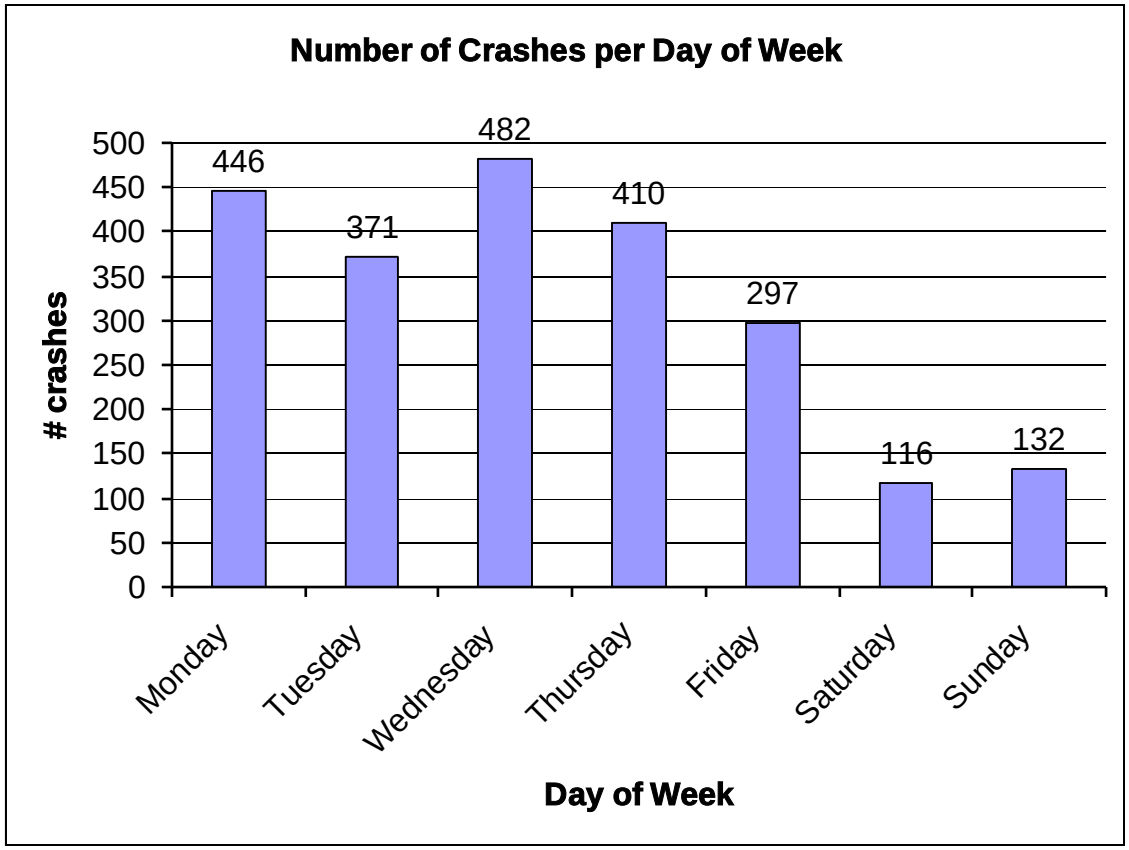

Figure 8: Crashes generated on each day of the week. This graph depicts the number of crashes that occurred on each day of the week based on the UC Berkeley EECS data set.

Studying Figure 9 and Figure 10, we observe an approximate correlation between usage and crashes during each hour of the 24-hour day. Most people work during the typical hours of 9am to 5pm. Since our data set involves users of various affiliations to the department, we see a wider spectrum of work schedules. While most administrative staff work during the day, several graduate students work in the evenings and late nights. Of course, this trend is likely to vary based on conference deadlines and course project deadlines. The crashes generated during each hour correlate fairly well with the usage for that hour of the day. Most crashes occurred between 1pm and 5pm, which is the same window for maximum computer usage in the department.

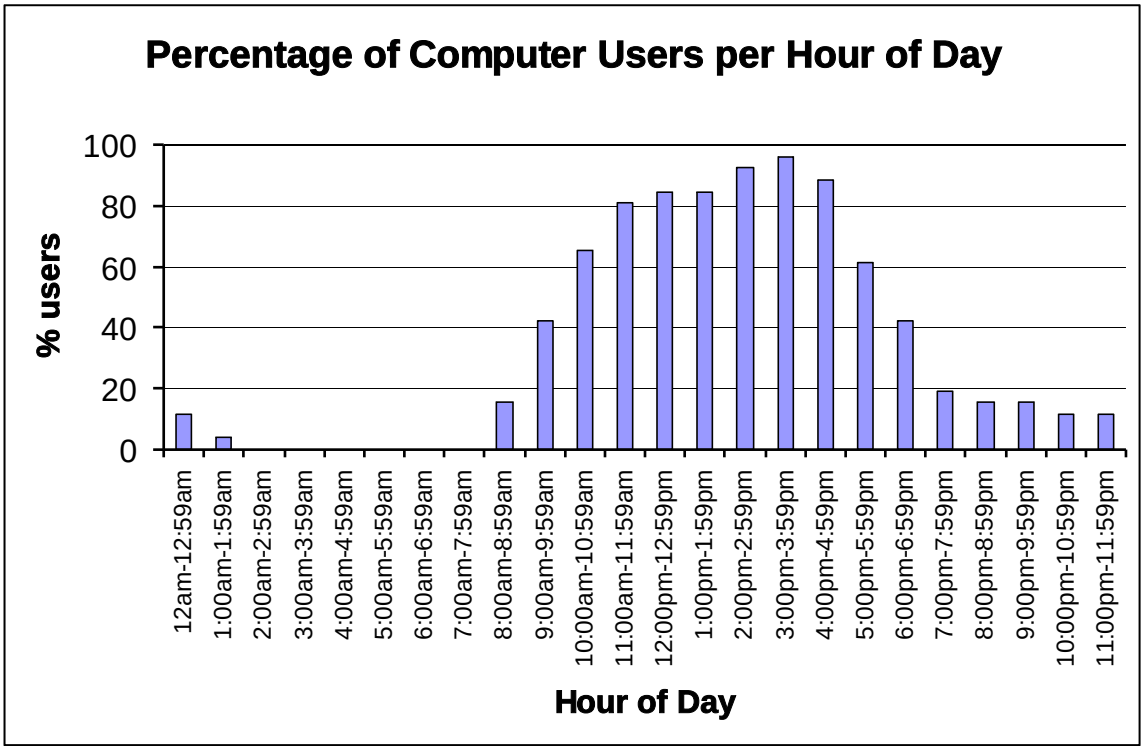

Figure 9: Computer Usage by Hour of Day. This graph depicts the percentage of our survey responders that use their EECS computer during each hour of the day.

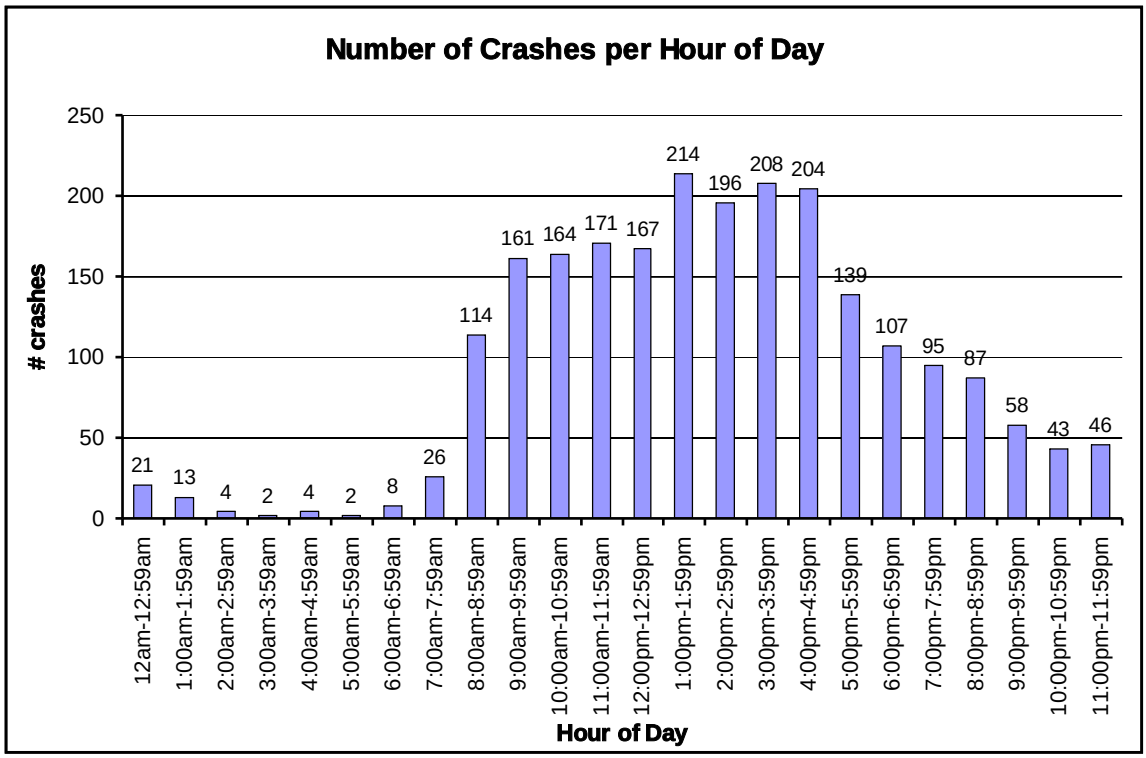

Figure 10: Crashes generated during each hour of the day. This graph depicts the number of crashes that occurred during each hour of the day based on the UC Berkeley EECS data set.

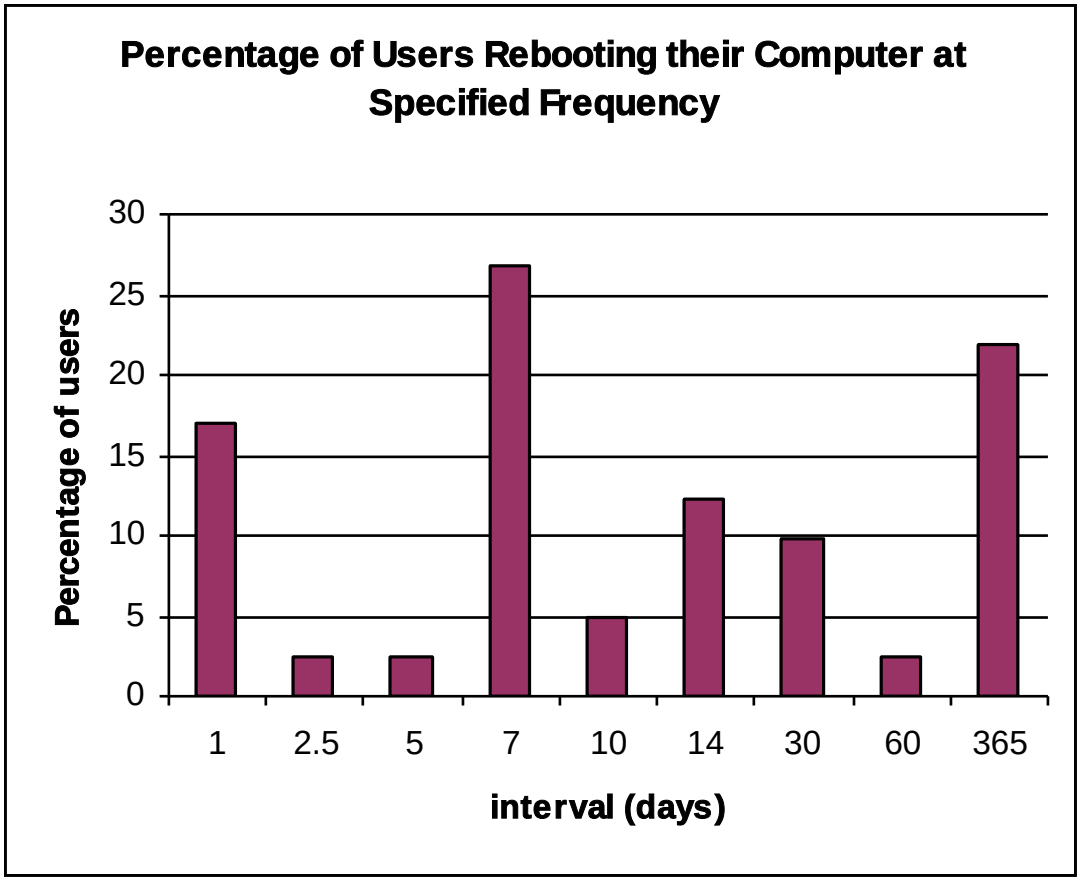

Figure 11: Frequency of Computer Reboot. This graph was generated based on responses to the survey we conducted among EECS department computer users whose machines contributed crashes to our data set.

In our survey, we also inquired about the frequency with which users reboot their machines. The rebooting process helps rejuvenate PCs and restore them to a clean and stable state. Based on the results in Figure 11, the reboot frequency largely varies among the users. Thus, it is difficult to generalize and derive conclusions on the quality of maintenance of the machines in the EECS department.

Windows users typically have several applications running in parallel; while many windows are open at a time, only one or two are actively used in the foreground, leaving other applications to run in the background. For example, instant messaging has become a common tool for communicating between friends, co-workers, and even meeting new people. Many users we asked use instant messaging software during their work hours. It is difficult to gauge the frequency of active use of such messaging software; however, the survey revealed that such software is definitely being used in the EECS department, justifying the handful of crashes generated by this category of applications.

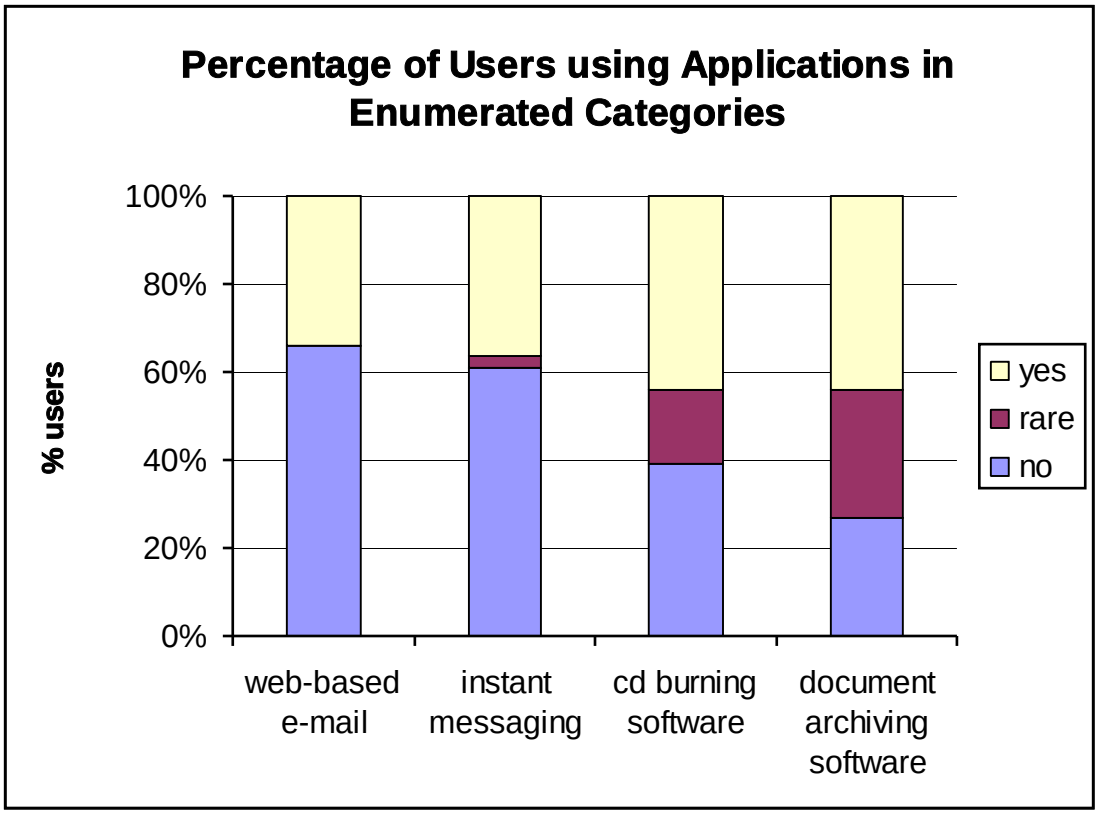

Figure 12: Usage frequency of miscellaneous applications

In our user survey, we also asked users to indicate if they use certain types of software (results in Figure 12). It is difficult to quantify the frequency of use of a category of applications if it is by nature useful on rare occasion. For example, cd burning software and document archiving software are not typically used on a regular basis. Thus, we wanted to verify that the lack of crashes in these categories are justified by the usage patterns. The only reliable source for understanding such usage patterns is to monitor processes on each computer.

7.1.3 Which Categories of Applications Generate the most Crashes?

Application crashes are more frequent than OS crashes but can usually be resolved by restarting the crashing application. Figure 13 shows a distribution of crashes by cause. Web browsers cause a majority of crashes in this dataset. This category includes Internet Explorer, Netscape, Mozilla and Firefox. One possible explanation for such a large number of browser crashes is that browsers interact with a wide variety of components which are often untested/unreliable. For example, people use web browsers for a variety of purposes including checking e-mail, interacting in chat rooms, uploading and downloading files and viewing multimedia. Plug-ins that are required to view a particular website often run inside browsers; crashes that were caused when interacting with a plug-in are blamed on the browser by the analysis tools.

The next major crash-contributing category is document preparation software. Applications in this category include MS Word, Powerpoint and LaTeX. According to the usage survey we conducted, on average, users spend 22% of their computer time

Application Category	# Crashes	Crash %	Usage %
web browsing	598	41%	18%
unknown	185	13%	n/a
document preparation	152	11%	22%
email	130	9%	24%
scientific computing	95	7%	7%
document viewer	84	6%	8%
multimedia	57	4%	6%
code development	26	2%	10%
document archiving	23	2%	n/a
remote connection	23	2%	n/a
instant messaging	17	1%	n/a
i/o	15	1%	n/a
other	14	1%	1%
database	8	1%	n/a
system management	8	1%	4%
security	7	0%	n/a

Figure 13: Crash Cause by Application Category. This table depicts the relative frequency of crashes caused by each category of applications and the relative time spent using each category of application (based on the user survey conducted in the EECS department at UC Berkeley).

using document preparation applications. It seems reasonable that such a highly utilized set of applications generate a large number of crashes. However, there are other factors than usage alone that must be addressed when justifying the crash rate. Document preparation software usually involves interacting with various data formats. For example, examining this thesis report alone, we find textual contents as well as Excel tables, graphs and pictures. Typically, document preparation software requires libraries to interact with a variety of data formats and this dependency results in increased number of crashes.

E-mail software such as MS Outlook and Eudora caused 9% of crashes in this data set but were reported to be used most frequently by users who responded to our survey. It is not surprising that e-mail generate so many crashes as there are numerous inconveniences that accompany e-mail. For example, numerous worms and viruses spread via e-mail, especially through e-mail attachments. Furthermore, some of the users we surveyed used e-mail programs that were embedded in their web browser (see Figure 12); however, crashes in such browser-related e-mail programs were classified as web browser crashes as we were unable to distinguish them from regular browser crashes.

Scientific computing and code development software also caused a sizeable amount of crashes in our data set. We expect this result to be atypical of an average PC user. Graduate student users in the UC Berkeley EECS department are more likely to develop software and use scientific computing tools such as Matlab and Mathematica more frequently than a typical PC user (or even a non-graduate student user such as a staff member in the EECS department).

We have a significant percentage of crashes attributed to un-classifiable applications, that is, we were unable to identify the purpose of the software, perhaps because they were custom-written and used by a single/small set of users. A few applications in this category had ambiguous names such as setup.exe, which could have belonged to one or more application categories. Since we have no method of tracing applications on the crashing machines, we refrained from forcing them into one of the above mentioned application categories.

7.1.4 Do Web Browser Usage Patterns Reflect Web Browser Crash Patterns?

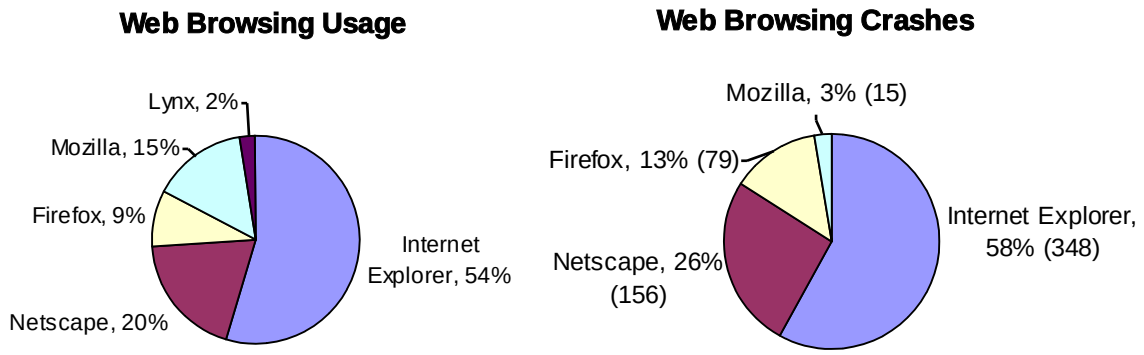

Figure 14: Web Browsing Application Use and Crash Frequency. Note: some users check e-mail using Netscape's built-in mail application. We do not distinguish between Netscape's browser and e-mail crashes.

While web browsers cause a majority of crashes in our data set, they are not the most frequently used application. We further dissected web browser crashes to identify the specific web browsing applications that contributed to these crashes (see Figure 14). Internet Explorer is the most commonly used, and the highest crash contributor among web browsers. Netscape and Firefox have approximately the same proportion of use as well as crashes. However, we must keep in mind that Netscape comes with a built-in e-mail client, that may have contributed some of the Netscape crashes in our data set. Mozilla also appears to be a fairly popular browser; however, it does not generate nearly as many crashes as other browsers. A possible explanation for Mozilla's robustness is that it is an open source product. Unlike proprietary software, Mozilla's code has benefited from thorough testing and evaluation from numerous users around the world. While Firefox is also open source, it is "younger" than Mozilla and consequently has less stable code that is more crash prone than Mozilla.

On average, users reported more frequent usage of email and document preparation applications than web browsers; these applications caused a significant proportion of crashes. Recall throughout this analysis that this data represents the Berkeley EECS department and not the entire Windows user population. Usage statistics underscore this fact as code development and scientific computation are uncommon activities for most Windows users.

7.1.5 What Causes these Crashes?

Figure 15 suggests that approximately half of crashes are generated due to a user's manual termination of an application, i.e., application hang. Often, when an application does not respond in a timely manner, perhaps due to an outdated .dll, an overloaded processor or insufficient memory, users tend to terminate this process and retry subsequently. It is possible that such applications would crash eventually if the user avoided pre-termination during its "hang". It is equally likely that the process was simply slow in responding and would have eventually completed the task successfully. Application hangs do not reveal much information regarding what occurred at the time of crash. Thus, we can not explore the details of such events.

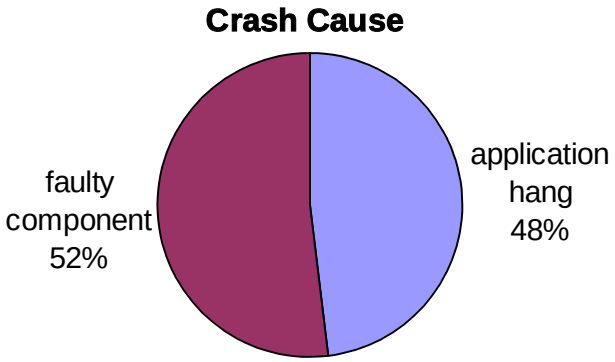

Figure 15: Crash Cause. This pie chart distinguishes the frequency of crashes due to application hangs from crashes caused by faulty components such as .dll, .exe and .sys files.

Application	# hangs	% hangs	% Running Total
iexplore.exe	185	25%	25%
matlab.exe	68	9%	34%
winword.exe	67	9%	43%
outlook.exe	60	8%	51%
firefox.exe	47	6%	57%
netscape.exe	41	6%	63%
unknown	25	3%	66%
powerarc.exe	19	3%	69%
powerpnt.exe	13	2%	71%
thunderbird.exe	13	2%	73%
excel.exe	12	2%	75%
acrobat.exe	11	1%	76%
explorer.exe	11	1%	77%
mozilla.exe	11	1%	78%
acrord32.exe	10	1%	79%
msimn.exe	10	1%	80%
AdDestroyer.exe	7	1%	81%
wmplayer.exe	7	1%	82%
notepad.exe	6	1%	83%
rundll32.exe	5	1%	84%
hp precisionscan pro.exe	4	1%	85%
mathematica.exe	4	1%	86%
msaccess.exe	4	1%	87%
msdev.exe	4	1%	88%
photosle.exe	4	1%	89%
winamp.exe	4	1%	90%
apps causing <1% of crashes each	84	11%	101%
Total	**736**		

Figure 16: Frequency of Hangs due to Various Applications. Some of these applications are custom-authored by users. Percentages shown are rounded to the nearest percent, causing the total to exceed 100%.

Applications hang frequently

Figure 16 outlines the applications that commonly hang. Again, Internet Explorer, represents the largest proportion of applications that hang; Netscape and Firefox fall among the top ten commonly hanging applications. A feasible explanation for this trend is that web browsers interact with numerous other applications such as Macromedia Flash, Quicktime, and Acrobat Reader. Consequently, a robust browser application is forced to interact with other applications that may not be safeguarded against unreliable code. To resolve this problem, interaction must be restricted to trusted, safe plug-ins, avoiding potentially unsafe and potentially malicious code. In contrast, applications such as MS Word, Outlook and Matlab can hang for different reasons. Often, large computations running in Matlab can use a significant amount of memory and disable other applications from running in parallel. In addition to insufficient computation memory, corrupt files can also cause the application to hang. In some scenarios, a file can be large enough to cause problems at start up. A practical solution must reduce the workload and/or upgrade the software/machine.

Component	Description	Author	Apps invoking component	%crash
ntdll.dll	NT system functions	MS	Internet Explorer, Matlab	11% (86)
msvcrt.dll	Microsoft C runtime library	MS	Acrobat, Netscape	5% (37)
acrord32.exe	Acrobat Reader	3rd party	Acrobat Reader	4% (29)
pdm.dll	Scripting component functions	MS	Visual Studio, Internet Explorer	3% (23)
firefox.exe	Web browser	3rd party	Firefox	2% (19)
user32.dll	Communication, message handler, timer functions	MS	Firefox, Internet Explorer	2% (17)
ray_tracing.exe	User application	3rd party	--	2% (16)
winword.exe	Windows document editor	MS	Word, Outlook	2% (15)
mshtml.dll	HTML related functions	MS	Internet Explorer, Netscape	2% (15)
tempest.exe	Unknown	3rd party	--	2% (15)
gklayout.dll	Mozilla layout library	3rd party	Thunderbird, Firefox	2% (14)
kernel32.dll	Microsoft memory management, I/O and interrupts library	MS	Acrobat, Firefox, Internet Explorer	2% (14)
simpl_fox_gl.exe	User application	3rd party	--	2% (14)
rpcl3260.dll	Real Player component	3rd party	Real Player	2% (13)
thunderbird.exe	Mozilla e-mail program	3rd party	Thunderbird	2% (13)

Figure 17: Top fifteen problematic DLL and executable files causing crashes. Each component is annotated with a description of its functionality, authorship (MS=Microsoft) and examples of applications using this component. The percentage of crashes attributed to a component is listed in the last column along with the raw number of crashes in parenthesis. This percentage excludes crashes categorized as application hangs. For user-written executable files, we are unable to provide sample applications that use the component.

.dll files are not robust enough

Figure 17 lists the top fifteen .dll and executable files blamed for crashes. These components constitute a significant portion of non-application hang-induced crashes. Apparently, a majority of problematic .dll files are invoked by multiple applications. A few noteworthy examples are ntdll.dll and msvcrt.dll. Among several scenarios, the same .dll can be blamed for a crash. For example, the caller of a .dll routine can pass invalid arguments to the callee. Alternately, a .dll's callee routine can return a bad value. Moreover, it is possible for a machine's state to be corrupt at the time of .dll execution. Precise inter-.dll interface definition and sand-boxing will help avoid cascading effects of data corruption.

.dll "Model Citizens"

We further investigated which dll files are used commonly among 33 applications we examined. We were limited to these 33 applications as we did not have executable files readily available for other applications. We generated dll dependency graphs using Dependency Walker (available at http://www.dependencywalker.com/). We identified commonly used dlls that never crashed and those that generated many crashes. Among 33 applications for which we were able to generate dll dependency graphs, 227 unique dll files were used. Of these dlls, only 37 caused crashes in our data set. The worst offenders were widely used components that provide Windows operating system functions. See Figure 18 for the 5 most commonly used dlls that produce several crashes. The top offender, used by all 33 applications and generating 86 crashes in our data set is ntdll.dll. Perhaps sandboxing this dll better will eliminate many future crashes.

Crash-causing dll	Num Crashes
ntdll.dll	86
msvcrt.dll	37
user32.dll	17
mshtml.dll	15
kernel32.dll	14

Figure 18: Commonly used dlls that produce several crashes. This list is based on 33 applications that we analyzed.

The remaining 190 dlls are "model citizens" for good dll design and implementation. Of these 190, 96 dlls were used by 32 out of 33 applications analyzed. These top dlls are listed below in Figure 19. One explanation for the success of these dlls is that they provide a focused set of functions. For example, netman.dll is responsible for managing network connections. Perhaps the best model for a dll is that it provides a small set of specific functions and intensively checks parameters for invalid values, eliminating errors at the earliest point possible.

ACTIVEDS.DLL	IMAGEHLP.DLL	NETMAN.DLL	SECUR32.DLL
ADSLDPC.DLL	IMM32.DLL	NETPLWIZ.DLL	SETUPAPI.DLL
ADVAPI32.DLL	INETCOMM.DLL	NETRAP.DLL	SHLWAPI.DLL
ADVPACK.DLL	IPHLPAPI.DLL	NETSHELL.DLL	SHSVCS.DLL
ATL.DLL	IRPROPS.CPL	NETUI0.DLL	TAPI32.DLL
AUTHZ.DLL	LINKINFO.DLL	NETUI1.DLL	URLMON.DLL
CABINET.DLL	LZ32.DLL	NETUI2.DLL	USERENV.DLL
CDFVIEW.DLL	MFC42U.DLL	NTDSAPI.DLL	UTILDLL.DLL
CERTCLI.DLL	MLANG.DLL	NTLANMAN.DLL	VERSION.DLL
CFGMGR32.DLL	MOBSYNC.DLL	ODBC32.DLL	W32TOPL.DLL
CLUSAPI.DLL	MPR.DLL	OLEACC.DLL	WINMM.DLL
COMDLG32.DLL	MPRAPI.DLL	OLEDLG.DLL	WINSCARD.DLL
CREDUI.DLL	MPRUI.DLL	OLEPRO32.DLL	WINSPOOL.DRV
CRYPT32.DLL	MSASN1.DLL	POWRPROF.DLL	WINSTA.DLL
CRYPTUI.DLL	MSGINA.DLL	PRINTUI.DLL	WINTRUST.DLL
CSCDLL.DLL	MSI.DLL	QUERY.DLL	WLDAP32.DLL
DBGHELP.DLL	MSIMG32.DLL	RASAPI32.DLL	WMI.DLL
DEVMGR.DLL	MSOERT2.DLL	RASDLG.DLL	WS2_32.DLL
DHCPCSVC.DLL	MSRATING.DLL	RASMAN.DLL	WS2HELP.DLL
DNSAPI.DLL	MSSIGN32.DLL	REGAPI.DLL	WSOCK32.DLL
DUSER.DLL	MSVCP60.DLL	RPCRT4.DLL	WTSAPI32.DLL
EFSADU.DLL	MSWSOCK.DLL	RTUTILS.DLL	WZCDLG.DLL
ESENT.DLL	NETAPI32.DLL	SAMLIB.DLL	WZCSAPI.DLL
GDIPLUS.DLL	NETCFGX.DLL	SCECLI.DLL	WZCSVC.DLL

Figure 19: Most commonly used dlls that do not crash. This list is based on 33 applications that we analyzed

Crash-causing Status Codes

The table in Figure 20 shows a list of the error codes that accompanied each application crash in our data set. Most of these errors (such as access violation) are essentially due to bad pointers. A significant number of crashes could have been avoided if processes stayed within their bounds and did not try to access memory that they did not have permission to use. A meta-lesson is that the code can immensely benefit from more careful boundary checking and verification.

The next highest crash-causing error code, 0xcfffffff, suggests an application hang. As mentioned earlier, hangs are a result of users manually terminating a non-responding application, which may potentially respond given sufficient time. We do not have enough information regarding application hangs to suggest techniques to avoid them. In page errors occur when an I/O request was incomplete and consequently, the contents were not appropriately loaded in memory. There are also a handful of exceptions due to integer and/or floating point arithmetic that was illegal or caused an overflow. These exceptions are often techniques to check for corner-case errors and can be prevented only by fixing the code that led to the corner-case. Some exceptions are due to code that does not abide Windows NT specifications. For example, invalid lock sequence status is a result of bad lock ordering according to Windows NT standards.

NTSTATUS code	Error Message	Num Crashes
0xc0000005	STATUS_ACCESS_VIOLATION	728
0xcfffffff	HANG	579
0xc0000006	STATUS_IN_PAGE_ERROR	15
0xc0000096	STATUS_PRIVILEGED_INSTRUCTION	11
0xeedfade	Trappable error in external object	7
0x80000003	STATUS_BREAKPOINT	6
0xc000001d	STATUS_ILLEGAL_INSTRUCTION	4
0xc0000409	STATUS_STACK_BUFFER_OVERRUN	4
0xc0000094	STATUS_INTEGER_DIVIDE_BY_ZERO	3
0xc0000025	STATUS_NONCONTINUABLE_EXCEPTION	2
0xc0000091	STATUS_FLOAT_OVERFLOW	2
0xc0150010	STATUS_SXS_INVALID_DEACTIVATION	2
0xe06d7363	Trappable error in external object	2
0xc000001e	STATUS_INVALID_LOCK_SEQUENCE	1
0xc0000090	STATUS_FLOAT_INVALID_OPERATION	1
0xc00000fd	STATUS_STACK_OVERFLOW	1
0xc015000f	STATUS_SXS_EARLY_DEACTIVATION	1

Figure 20: Crash-causing status codes. Status codes were available for only a subset of the crashes.

7.2 OS Crashes

OS crashes are more frustrating than application crashes as they require the user to kill and restart the explorer process at a minimum, more commonly forcing a full machine reboot. While there are a handful of crashes due to memory corruption and other common systems problems, a majority of these OS crashes are caused by device drivers (as seen in Figure 21). These drivers were related to various components such as display monitors, network and video cards.

7.2.1 What are Device Drivers?

A device driver is a kernel-mode module that communicates operating system requests to the device and vice versa. These drivers are inherently complex in nature and consequently difficult to write. Among many reasons for device driver complexity are that these drivers deal with asynchronous events. Since they interact heavily with the operating system, the code must follow kernel programming etiquette (which is difficult to master and follow). Furthermore, once device drivers are written, they are exceedingly difficult to debug as the typical device driver failure is a combination of an OS event and a device problem, and thus very difficult to reproduce (see [SM+04] for a detailed description of device driver problems).

Figure 21, in addition to pointing out the high number of device crashes, also specifically shows the number of graphics driver crashes. The fields for this table were a direct result of scraping the OS Crash Type field from all the analyzed crash dumps. For legal reasons, the publicly available analysis tools do not reveal driver categories for various crashing drivers. Graphics drivers appear to be an exception to the rule.

OS Crash Type	Num Crashes
DRIVER_FAULT	458
COMMON_SYSTEM_FAULT	63
GRAPHICS_DRIVER_FAULT	36

Figure 21: Number of OS crashes of each type. This table was generated based on the OS Crash Type field in analyzed crash reports.

7.2.2 What Components Cause OS Crashes?

In the absence of more details revealed by the analysis tools, we considered guessing the type of each driver that caused a crash. However, we realized this effort might lead to inaccurate results and numerous unknown mappings. There are thousands of drivers available and not all of them have English documentation. Thus, it would take a large amount of effort to web crawl and gather data, and perhaps not embellish this work too much more. As an alternative, we study the image (i.e. .exe, .SYS or .dll file) that caused these crashes, so we can at least identify the organization that contributed the crash-causing code (see Figure 22).

Image Name/ Crash Cause	Image Description	Num crashes	% crashes	% Running Total
ntoskrnl.exe	NT kernel and system	150	27%	27%
GDFSHK.SYS	McAfee Privacy Service File Guardian	42	8%	35%
ALCXWDM.SYS	Windows (R) WDM driver for Realtek AC'97	40	7%	42%
kmixer.sys	kernel audio mixer of Microsoft Windows	28	5%	47%
win32k.sys	multi user win32 driver	19	3%	50%
ati3d2ag.dll	ATI Technologies Inc. Radeon DirectX Universal Driver	18	3%	53%
Brwgate.sys	NAT/Proxy/Firewall system	16	3%	56%
HSF_CNXT.sys	Conexant Systems SoftK or SoftK56 Modem Driver	10	2%	58%
Ialmdev5.DLL	Intel graphics driver	10	2%	60%
ati2dvag.dll	ATI Radeon WinNT display driver	8	1%	61%
nv4_disp.dll	NVIDIA Compatible Windows 2000 display driver	8	1%	62%
V7.SYS	IBM V7 Driver for Windows NT/2000	8	1%	63%
usbscan.sys	Microsoft usb driver	7	1%	64%
ALCXSENS.SYS	Windows (R) WDM driver for Realtek AC'97	6	1%	65%
ar5211.sys	driver for dual band WIFI wireless mini pci adapter	6	1%	66%
pcx500.sys	NDIS5.1 Miniport Driver for 32 bit Windows	6	1%	67%
Unknown_Image	--	6	1%	68%
ati3duag.dll	ATI Technologies Inc. Radeon DirectX Universal Driver	5	1%	69%
AVGNTDD.SYS	Filter Device for Windows XP/2000/NT	5	1%	70%
nv4_mini.sys	NVIDIA Compatible Windows 2000 Miniport Driver	5	1%	71%

Figure 22: Top 20 OS Crash-causing Images. A description of the crash-causing image is provided in addition to the percentage of crashes caused by each image.

The top contender in Figure 22 is ntoskrnl.exe, which constitutes the bare-bones Windows NT operating system kernel code. It is not surprising that this executable is responsible for a number of driver crashes because it interacts with every other operating system component and is thus the single most critical component that can never be perfect enough. Furthermore, other systems code might generate bad input parameters to the ntoskrnl functions that cause exceptions; ntoskrnl bears the blame for the resulting crash as it generated the exception.

Other crash causing images range from graphics drivers to multimedia and I/O drivers. It is difficult to debug or even analyze these crashes further as we do not have the code and/or symbols for these drivers. With the increasing need for numerous devices accompanying the PC, it does not scale for the operating system developers to account for and write device driver code for each device; consequently, device drivers are written by device manufacturers, who are typically inexperienced in kernel programming. Perhaps such lack of expertise is the most impacting cause for driver-related OS crashes.

We also had 47 OS crashes caused by memory corruption. Memory corruption-related crashes can often be attributed to hardware problems introduced by the type of memory used (eg. non-ECC memory). In the event that the memory corruption was due to software, the probelm cannot be tracked down to a single image.

Driver Fault Type	Num Crashes
PAGE FAULT IN NONPAGED AREA	118
IRQL NOT LESS OR EQUAL	105
KERNEL MODE EXCEPTION NOT HANDLED	67
UNEXPECTED KERNEL MODE TRAP	63
BAD POOL CALLER	46
THREAD STUCK IN DEVICE DRIVER	36
SYSTEM THREAD EXCEPTION NOT HANDLED	29
Unknown bugcheck code	16
Other (each caused 1 crash)	14
PFN LIST CORRUPT	13
DRIVER CORRUPTED EXPOOL	12
DRIVER UNLOADED WITHOUT CANCELLING PENDING OPERATIONS	8
MANUALLY INITIATED CRASH	5
File Corruption - Unreadable File	4
BAD POOL HEADER	4
KERNEL DATA INPAGE ERROR	4
NTFS FILE SYSTEM	4
CRITICAL OBJECT TERMINATION	3
FAT FILE SYSTEM	3
DRIVER POWER STATE FAILURE	2
KERNEL STACK INPAGE ERROR	2
MEMORY MANAGEMENT	2
MULTIPLE IRP COMPLETE REQUESTS	2

Figure 23: Crash generating driver fault type.

7.2.3 Which Faults Generate the Most OS Crashes?

To further understand driver crashes, we studied the type of fault that resulted in the crash. Figure 23 lists the number of crashes that were caused by the various fault types. These fault types are reported by Microsoft's analysis tools when analyzing each OS crash dump.

While many of these fault types are straightforward to understand from the name, many others are abbreviations of the event they describe. Below, we enumerate each fault type and its significance (based on the descriptions provided in the parsed crash dumps):

- **PAGE FAULT IN NONPAGED AREA** - Invalid system memory was referenced. This cannot be protected by try-except, it must be protected by a Probe. This error is typically due to a bad pointer. This category of driver faults contributed the most OS crashes in our data set.
- **IRQL NOT LESS OR EQUAL** - An attempt was made to access a pageable (or completely invalid) address at an interrupt request level (IRQL) that is too high. This is usually caused by drivers using improper addresses.
  *Note: The interrupt request level is the hardware priority level at which a given kernel-mode routine runs, masking off interrupts with an equivalent or lower IRQL on the processor. A routine can be preempted by an interrupt with a higher IRQL.
- **KERNEL MODE EXCEPTION NOT HANDLED** - The exception address pinpoints the driver/function that caused the problem. This address, combined with the date link date of the driver/image containing this address, can provide insight to the problem.
- **UNEXPECTED KERNEL MODE TRAP** - A trap occurred in kernel mode, either because the kernel is not allowed to have/catch (bound trap) the trap or because a double fault occurred.
- **BAD POOL CALLER** - The current thread is making a bad pool request. Typically this is at a bad IRQL level or double freeing the same allocation, etc.
- **THREAD STUCK IN DEVICE DRIVER** - The device driver is spinning in an infinite loop, most likely waiting for hardware to become idle. This usually indicates problem with the hardware itself or with the device driver programming the hardware incorrectly.
- **SYSTEM THREAD EXCEPTION NOT HANDLED** – This fault type is similar to an unhandled kernel mode exception. The exception address pinpoints the driver/function that caused the problem. This address, combined with the date link date of the driver/image containing this address, can provide insight to the problem.
- **PFN LIST CORRUPT** - Typically caused by drivers passing bad memory descriptor lists.
- **DRIVER CORRUPTED EXPOOL** - An attempt was made to access a pageable (or completely invalid) address at an interrupt request level (IRQL) that is too high. This fault is caused by drivers that have corrupted the system pool.
- **DRIVER UNLOADED WITHOUT CANCELLING PENDING OPERATIONS** - A driver unloaded without canceling timers, DPCs, worker threads, etc. The broken driver's name is displayed on the screen.

- **MANUALLY INITIATED CRASH** - The user manually initiated this crash dump. This is not really a problem, perhaps a test to verify that crash reporting works properly.
- **BAD POOL HEADER** - The pool is already corrupt at the time of the current request. This may or may not be due to the caller. The internal pool links must be walked to figure out a possible cause of the problem, and then special pool applied to the suspect tags or the driver verifier to a suspect driver.
- **KERNEL DATA INPAGE ERROR** - The requested page of kernel data could not be read in. This fault is typically caused by a bad block in the paging file or disk controller error. Possible errors include a failure experienced by the disk subsystem and a failed request due to a filesystem not progressing forward.
- **NTFS FILE SYSTEM** - This fault suggests a problem with the machine's NTFS filesystem.
- **CRITICAL OBJECT TERMINATION** - A process or thread crucial to system operation has unexpectedly exited or been terminated. Several processes and threads are necessary for the operation of the system; when they are terminated (for any reason), the system can no longer function.
- **FAT FILE SYSTEM** – This fault suggests a problem with the FAT filesystem on the machine.
- **DRIVER POWER STATE FAILURE** - A driver is causing an inconsistent power state.
- **KERNEL STACK INPAGE ERROR** - The requested page of kernel data could not be read in. This fault is typically caused by a bad block in paging file or disk controller error.
- **MEMORY MANAGEMENT** – Bad input parameter.
- **MULTIPLE IRP COMPLETE REQUESTS** - A driver has requested that an IRP be completed, but the packet has already been completed. In the best case scenario of this fault, a driver attempted to complete its own packet twice. More commonly, two separate drivers attempt to complete the same packed as each driver believes it owns the packet.

Studying these fault types reveals various programming errors that impact system behavior and what OS problems to tackle with caution. We explore the possibilities for improving PC reliability and evaluate their pros and cons in the next section.

7.3 Practical techniques to reduce crashes

In the past, Windows code has been optimized for performance rather than reliability. Much of the parameter checking code was eliminated in the interest for faster response. Currently, speed and performance are becoming less important than reliability. Perhaps it is time to reintroduce more parameter checking and verification at each system procedure call interface.

Traditionally, software reliability problems have been tackled by using a handful of ad-hoc methods. The ideal solution for increasing software reliability is to work with existing components and simply add wrappers/helper components to help the software

function better and/or recover gracefully. However, this solution is not always an option. An alternative technique is to rewrite the unreliable code. Unfortunately, this approach does not scale well, especially with thousands of proprietary software running on PCs. One can also rebuild the entire system from scratch – redesign the Windows operating system, redefine interfaces and standards for third party application and device driver code. This approach, while feasible, is extremely unrealistic as the cost-benefit analysis would reveal that the cost would far outweigh the benefits. Moreover, there is no guarantee that everyone will converge on these new standards, let alone guarantee status-quo reliability.

Software-Based Fault Isolation

Wahbe et al. [WL+93] proposed isolating distrusted modules from trusted OS components by loading code and data into its own *fault domain.* Such sand-boxing would enforce clear semantics for code/data flow between trusted and untrusted modules. It would allow each kernel extension to execute independently of other extensions while having access to a specified portion of kernel memory. The authors suggest using cross-fault-domain RPC to invoke code or modify data and maintain dedicated arbitration code (in its own fault domain) to decide if the cross-fault-domain transactions are safe.

This model would be very valuable in the Windows operating system. Clearly, we could reduce the number of OS crashes caused by bad device driver code using this fault isolation technique. We could treat all third party device driver code as "untrusted" and execute each of them in its own fault domain. This model guarantees that an "untrusted" device driver could not clobber operating system state. The crucial component of this model is developing the arbitration code; incompetent arbitration code is at least as bad as not having any isolation at all. This approach requires modifying existing operating system code to incorporate the notion of fault domains. However, with the size and complexity of Windows operating system code, this option might not be feasible in a short time span.

Nooks

Nooks, discussed in [SM+04], tackles the device-driver unreliability problem by adding a subsystem in the Linux kernel to handle reliable device-driver communications. Clearly, mechanisms implemented by Nooks would help reduce device driver-related crashes. They follow the software fault isolation model by enclosing each kernel extension in a dedicated protection domain. All communication between the kernel and its extensions passes through a wrapper that enforces the use of extension procedure calls (XPC). To allow graceful recovery, they implement shadow drivers that monitor each driver and take over in the case of a failure. A functional shadow driver enables the original (faulting) driver to be reset, allowing relatively transparent failure recovery from the driver fault.

While this mechanism is effective in preventing the propagation of driver faults to the user level, it seems difficult to scale to all possible driver categories. For example, Nooks has been tested on a handful of device categories (such as network cards and sound cards), and the communication between these drivers and the operating system have been successfully sand-boxed to allow monitoring and post-mortem analysis. Furthermore, it seems as though Nooks works best if the kernel extensions can be

terminated and restarted safely; it is unclear how the model would change if safe terminations and restarts were not the norm.

Separate protection level for drivers

The MULTICS operating system adopted multiple protection rings for restricting the flexibility of applications [SS72]. In most modern operating systems (including Windows), we simply distinguish between kernel and user level permissions (effectively 2 protection levels rather than 9 prescribed by MULTICS). With un-trusted driver code requiring access to communicate with the operating system, it is obvious the device drivers should have more flexibility than application code that executes in the users level. However, it seems unnecessary for device drivers to have complete flexibility to modify and often corrupt operating system structures, especially when the code is written mostly by third party vendors who are not familiar with the detailed workings of the operating system.

Perhaps it is wise to introduce an intermediate protection level for device drivers as a compromise between user level restrictions and kernel level freedom. Obviously, this approach would require clear redefinition of interfaces between the kernel level and the intermediate protection level. It also requires rewriting portions of the operating system code and moving existing kernel-level device driver code into this new intermediate protection level.

Move driver code to user level libraries

Along the lines of the previously proposed solution of creating a new protection level, we can perhaps consider moving all driver code entirely to the user-level. This modification would completely restrict the amount of "damage" driver code can do to the operating system. Perhaps we can create user-level libraries that interface with the operating system and validate communications between device drivers and the operating system. This approach involves moving some of the operating system code (and all device driver code) into the user-level.

Virtual Machines for unsafe/distrusted applications

Virtual machines offer a mechanism for isolating the effects of one application from another. Recently, there have been many opportunities to use virtual machines for improving the reliability of operating systems. In [KD+04] the authors run code in virtual machines and log/monitor operations on the host machine to track and understand the interactions between the various applications and the operating system. Furthermore, with the increase in e-mail viruses, several PC users open their e-mail applications in a virtual machine so that any mal-effects of e-mail are contained within that virtual machine and do not affect other applications on the host machine.

We can take advantage of virtual machine technology to reduce the number of crashes on PCs. Currently, there is no mechanism to transparently invoke virtual machines upon application start. We can run unreliable/distrusted applications on these dedicated virtual machines to understand their impact on other concurrently executing applications. Over time, if an application has functioned without causing any unexpected behavior, we can migrate the application to the host machine. This area of research sounds attractive as it does not require rewriting operating system or application code.

Upon application invocation, we simply need to verify the integrity/trustworthiness of applications and select between running them on the host machine or a dedicated virtual machine.

This technique would be particularly useful for crashes caused by web browsers (a majority of application crashes). Since web browsers often invoke multiple “helper” components such as plug-ins, invoking a web browser in a separate virtual machine ensures that the plug-ins are also invoked in the same virtual machine and do not affect any other applications running on the bare machine. Crashing the virtual machine is less problematic than crashing the host machine as the number of peer applications affected by the crash are drastically reduced.

8. Discussion – A Case for an Open Source Data Repository

An Open Source Data Repository would simplify data collection and make failure data more accessible to systems researchers. We make a case for such a repository by identifying the drawbacks of our current data collection and analysis techniques, and suggest issues to consider when designing such a repository.

8.1 Drawbacks of Current Data Collection Mechanisms

In this section, we discuss several obstacles and consequent shortcomings of our data analysis and ways to address them.

8.1.1 Insufficient Data Quantity

One can think about application complexity and crash-susceptibility in terms of interface complexity. The interfaces between an executable and its libraries, between binary files and the system configuration, and between the user and the application all introduce complexity, and are easier to quantify than the source-code complexity of an application with a range of external dependencies. To address each of these interfaces, we would like our data set to include multiple crash events, occurring on different computers with different usage patterns, for each of the (application, DLL) pairs occurring most frequently in home or corporate settings. For the 33 applications we could analyze, we found 227 DLLs in use. Expanding this list to 50 applications and assuming around 250 DLLs in use, this comes to around 12,500 (application, DLL) pairs.

This "back-of-the-envelope" calculation provides a foothold for an order-of-magnitude estimate of the sample size that might provide reliable clustering data. If we let the number of (application, DLL) pairs approximate the number of outcomes we care about (the interaction between (application, DLL) pairs and error codes is difficult to quantify because the two are not causally independent), then the multinomial distribution of failures given a machine configuration and usage pattern has 12,500 outcomes. It is difficult (although not entirely impossible) for a researcher to single-handedly collect and analyze so many different outcomes. However, Microsoft has the resources and the data (and access to the source code) to investigate all these possibilities. In our case, the range of potential causes (corrupted DLLs, version conflicts, misconfiguration, and user behavior, among others) only serves to enlarge the space of possible outcomes. We will require more data to make more concrete claims about the results we observed.

8.1.2 Improving BOINC Data Quality

Ideally, to embellish our analysis with information about the sequence of events and/or the machine's condition leading to the crash, we wish to know precisely the duration of each application or process and the associated resource consumption. A continuous profile of the machine's evolution is absent in the collected data. For each machine, it is useful to know information including service packs, CPU type/speed, RAM, disk capacity, applications installed, antivirus tools installed, virus definition date/version. We

must also collect several performance metrics, expressly before and during the crash. For example, for each machine, it is useful to know the system uptime, amount of free space, number of processors, processor queue length(s) and network configurations. Such data can suggest the sequence of events that lead to a crash and factors and processes that influence the failure progression. Presently, as we rely on Microsoft's debugging tools to parse crash dumps, it is difficult to study the context of each failure as third party executable images are encoded not to be publicly available. Collecting machine metrics and process information will improve the accuracy of our analysis process.

8.1.3 Difficulty of Collecting Data

Several limitations are imposed on our analysis due to the inherent concern regarding privacy. We have observed that some people are undoubtedly uncomfortable with data collection. After crossing the initial threshold to gain credibility, people are eager to share crash data. The privacy issue is a matter of policy rather than data availability. The most outstanding concern that users have with sharing data is anonymity. It is crucial for us to mask the exact data source and collect and store data anonymously. Thus, data collection must be restricted to necessary and sufficient statistics that evaluate usage. Also, it is beneficial to provide incentives to the users volunteering their data. This technique has been fruitful in past projects and continues to be an attractive mechanism to gather data.

Usage data collection continues to be difficult, even in the UC Berkeley EECS department. People are concerned that revealing usage information allows others to reverse-engineer data and hold potentially incriminating evidence against the user. In contrast, the computer industry routinely collects such data and is keen to share this information. Perhaps businesses expect a pattern of usage behavior from their employees and are less concerned with privacy.

A major bottleneck with industrial collaborations is legal documentation. Based on our experience, engineers are willing to share the company's data if they see an incentive such as being able to know what applications to avoid using. Corporate lawyers, who draft agreements for the collaborations, are less willing to give us access to the data. It takes up to several months for them to draft a non-disclosure agreement, even if we already have agreements in place for other project collaborations with that company. Valuable time is lost during the interim period between submitting a request to the lawyer and getting signatures on the agreements. That time could have been used to collect more data and/or analyze and take the next steps in making changes based on the analysis results.

8.2 Design Challenges for an Open Source Data Repository

Gathering data requires a significant investment. Some of these investments are technical, such as the cost of building an infrastructure for measurement. Others are more social in nature. For example, when collecting data from companies, researchers must pass through several layers of indirection for an approving signature. Furthermore, corporate lawyers spend several months drafting tedious legal agreements for the collaboration. Similarly, when conducting user studies for research, students are often required to obtain

approval from an institutional review board, which requires considerable paperwork and has high latency.

It is accepted wisdom that time is our most precious resource. It can take days, weeks, or months to collect a large quantity of useful data. Nevertheless, we often fail to amortize these costs and leverage these investments. If we are to be more effective and efficient as a research community, then we must find ways to use and build upon the time investments of our peers. Data collection in experimental computer science and engineering has always been time consuming. However, as we pursue research agendas which increasingly connect the physical and virtual worlds, or embark on projects which require data as an input, like applications of statistical learning theory to systems problems, the data collection, management, and sharing challenges can only grow.

There is currently no single repository that can be queried for data sets. We rely on the knowledge of peer researchers to point us to the right person to obtain data. An open source data repository provides a single interface to numerous data sets, eliminating the unnecessary downtime of waiting for responses. There are various design and maintenance considerations for building such a repository, some of which are enumerated below.

One of the biggest challenges of building an open source data repository is determining management logistics. A centralized repository would be simpler and more cost effective to monitor. Designating a single organization to maintain the system introduces issues related to economics as well as trust. A decentralized repository, on the other hand, would be more fault tolerant (eliminating the single point of failure) but would require sophisticated consistency mechanisms to assure data integrity. A related question is whether federated management is feasible; a federated scheme would allow member sites to choose which features to open or not to the outside.

There are numerous repositories created by research groups to hold different types of data ranging from failure data to http and Apache logs to sensor data. Little effort is spent on making these repositories easy to access. Systems researchers would benefit from a unifying schema that accommodates all these data types. We can use XML-like languages to write headers describing the data set. Given such data descriptor headers, we can provide tools to automatically convert the data to our desired format and store the information in our repository.

It is important to verify the authenticity of data (and the contributing entity) to avoid plaguing our repository with fake data. We also need mechanisms to verify that people using the data give due credit to the data contributors. This task is challenging as the purpose of the repository is to provide data access to any and all organizations while reducing the likeliness of misuse. Also, people often have stringent privacy requirements for sharing data. We can meet these requirements by providing an infrastructure to anonymize sensitive data at the point of collection, a model that has already been adopted by some systems researchers. Another concern is that no details regarding individuals should be reproducible from cross-correlating various data sets.

9. Conclusions

Our crash-data related study has contributed several Windows related revelations. The most notable reality is that the Windows operating system is not responsible for a majority of PC crashes at Berkeley. Application software, especially browsers, is mostly responsible for these crashes. Users can alleviate computer frustration by better usage discipline and avoiding unsafe applications. With additional data collection and mining, we hope to make stronger claims about applications and also extract safe product design and usage methodology that apply universally to all operating systems. Eventually, this research can gauge product as well as usage evolution.

Our study of operating system-level driver crashes has also revealed many insights. It is clear that PCs would benefit from enclosing device drivers in a more restrictive environment. Furthermore, better programming etiquette can avoid many problems introduced in device driver code. Most authors of device drivers are not trained sufficiently to follow kernel programming rules and best practices. It is perhaps time to offer mandatory training for these device driver authors and also develop better tools that advise and constrain device driver code.

The analysis performed in this report, if applied to a data set representative of Windows users world-wide, can help us derive conditions for safe and unsafe application functionality. We would be able to devise a knowledge-base of universally safe application configurations that would never crash as well as combinations of applications (and their versions) that are guaranteed to be problematic. Such information would forewarn users before they purchase software and also allow them to choose the level of risk they are willing to take for their personal computing experience.

Studying failure data is as important to the computing industry as it is to consumers. Product dependability evaluations, such as reports provided by J.D. Power and Associates, help evolve the industry by reducing quality differential between various products. Once product reliability data is publicized, users will use such information to guide their purchasing decisions and usage patterns. Consequently, product developers will react defensively to resulting competition. Perhaps using the data in this report, manufacturers of both hardware and software would pay considerable attention to their products thereby improving their quality control.

Appendix A: Usage Survey

As part of Prof. David Patterson's research on Recovery Oriented Computing, we are studying the cause of crashes on Windows PCs. To improve our analysis, we would appreciate if you could answer the questions below. While the real reward is societal, benefiting future generations of computer users, we will select 4 winners to receive a $50 gift certificate to Amazon.com. Please send questions/comments and survey responses to archanag@cs.berkeley.edu

---

1) On average, how many hours a day do you spend actively working on your EECS computer?

2) What are your usual hours of computer work on an EECS machine? (e.g. 9am-6pm, 3pm-1am, midnight-10am)

3) what days of the week do you usually use the EECS machine? (e.g. Mon-Fri, Wed-Sat, Thu-Mon, ...)

4) What percent of this time is spent actively on the following activities (and which software do you use):
Web browsing (internet explorer, netscape, ...)
Email (outlook, eudora, ...)
Document/presentation preparation (Word, Powerpoint, Latex, ...)
Document viewing (acrobat, ...)
Code development (C, C++, Java, visual studio, ...)
Scientific computation (matlab, mathematica, ...)
System management/security (e.g. install/uninstall software, antivirus, antispyware, ...)
Multimedia (Media player, Quicktime, ...)
Other (please specify)

5) Please answer yes/no to the following questions:
-is your e-mail program part of a web browser (e.g. netscape mail) Note: this does not include using web interface for e-mail?
-do you use instant messaging?
-do you use cd burning software?
-do you spend time compressing and uncompressing documents?

6) Which apps crash most frequently?

7) When an app crashes, what do you tyically do?
Restart app?
Restart computer?
Other ...

8) How frequently do you reboot your computer?

Appendix B: Clustering Windows Applications based on Crash Behavior

Automated techniques allow us to process larger data sets and identify deeper interactions between applications and shared libraries than approaches like manually partitioning crashes by publisher or task. However, automated clustering can only succeed if the data actually encode the characteristics relevant to the clustering task. This is especially challenging when the data set contains a wide range of non-numeric features, as is the case with the crash events we recorded. We will discuss the measures we took to augment crash data with relevant application, library, and workstation features; define distance measures appropriate to the structure we intended to capture, and ascertain the sensitivity of our model to changes in those distance measures.

We would like to use our automated clustering procedures to identify not only whether a particular deployment of an application to a workstation interacted poorly with the environment provided by that workstation in the past, but what underlying features of a program make it susceptible to failure so that future designs and implementation do not perpetuate those features.

Distance Measures for Crash-Event Clustering

Crash event vectors, including the raw data returned by Microsoft's CER and the derived features we append, mix several data types. Strings identify applications, libraries, users, and machines; hexadecimal values correspond to error codes returned by DLLs, and version numbers (integer arrays of length four) distinguish implementations of executables and DLLs with the same names. To these data we add set-valued (the DLL support of an application) and decimal-valued features (normalized times derived from the timestamp and self-reported usage frequencies).

A single similarity measure will not suit all of these data types. We could order string identifiers alphabetically, and measure their "similarity" as the edit distance between them, but this does not correspond to the similarities or differences in the applications or machines identified by those strings. (In more rigidly-administered environments, one can imagine using machine names to identify the deployment configuration of a computer, but our data suggest that this approach has little traction in the EECS department.) Likewise, the total ordering of timestamps does provide information about crash chains, but a total ordering of hexadecimal error codes is meaningless, since the error codes are (in general) arbitrarily assigned to error conditions. However, commonly-used metrics like the Euclidean distance between two vectors only depend on the *componentwise* difference between those vectors, and do not require that the same notion of "difference" apply to all components. This allows us to define a difference operation for each data type that is consistent with the clustering task at hand:

- *String identifiers* and *hexadecimal error codes* used a "binary difference" operator: if two values were equal, their difference was 0; otherwise, their difference was 1.
- *Decimal values* used ordinary subtraction; the magnitude of the difference between two values was as important as the existence of the difference.

- *Set-valued features* (like the DLL support of an application) used the size of the *symmetric difference* between the two sets. (The symmetric difference of two sets contains all elements that occur in exactly one of the sets.)
- *Version numbers*, logged as $v_1.v_2.v_3.v_4$, were compared one component at a time, and the differences summed. The difference between major versions v_1 could be arbitrarily large. The difference between minor versions v_2 was scaled over all pairs of version numbers observed to be less than 1.0, the difference between incremental updates v_3 was scaled to be less than 0.1, and the difference between builds was scaled to be less than 0.01. This corresponds to one interpretation of version-number semantics (that major version numbers provide the most useful measure of program complexity), but we discussed several others. For example, higher minor version numbers might correspond to bug-fix releases, or they may indicate the introduction of new features without the exhaustive testing that typically precedes a major release.

Normalization, Bias, and Default Values

With the diversity of data types came the challenge of scaling the feature space so that no one feature dominated the distance computation for a pair of data points. For example, if one measures the time between crash events in milliseconds, then the feature corresponding to the elapsed time since the last crash will overwhelm all other distance measures. On the other hand, if one measures time in days, the contribution of elapsed time to the distance between two data points is negligible

On the surface, normalization does not seem difficult: scale each component's difference operator so that the maximum difference between two components is 1. However, if all dimensions are scaled in this way, then the choice of features can materially alter the outcome of the clustering procedure. For example, we include both user name and machine name in our data set to ascertain the relative influence of user behavior and system configuration in the occurrence of crashes. However, machine names and user names are almost perfectly correlated in our data set; only one or two of the machines in our sample population generated crash reports with different users logged in. As a result, the identity of the user (or his computer) carries twice as much weight as the name of the application when deciding what crash events are related. Similarly, the limited number of crash events we collected shows a strong correlation between DLL-specific error codes and individual computers. Out of the nearly 1500 error reports, only two crash chains occurring on different machines share the same (DLL, error code) pair! This sparsity means that the introduction of derived features, rather than providing a richer set of data about the task domain for the clustering algorithm to arrive at a reliable conclusion, can serve as a tool to systematically bias the result. The verb-noun co-occurrence example mentioned above also ran that risk by replacing individual nouns with noun categories. However, the goal in that case was to compare the grammar-checker's internal representation of "common usage" against a large corpus of examples; the substitution of categories for individual nouns made for a tighter connection between the clustering results and the grammar-checker's representation. Our goals in clustering crash events are less concrete: we suspect that the application's intended purpose, its implementers' practices, the configuration and state of the host PC, and the sophistication of the user all play a role in the frequency and duration of crash chains, but we can't

	K-Means Clustering	AgglomerativeClustering
Initialization	Choose *k* data points uniformly at random from the full data set	Each data point is the center of a singleton cluster
Iteration	**1) Assign** each data point to the nearest cluster center **2) Recompute** cluster centers as the "average" of all member data points	**1) Merge** two clusters whose centers are closest to each other **2) Recompute** cluster centers as the "average" of all member data points
Runtime	$O(n^2)$	$O(n)$

Figure 24: A comparison of clustering algorithms.

provide additional information in any of those categories without running the risk that we privilege one over the others. At this stage of the work, we have taken our best effort to balance the feature set; the results described below seem to match our expectations. We see nothing to indicate that the features we have added greatly upset the balance of relevant characteristics.

The choice of default values provides another avenue for introducing bias. Both of the examples cited at the beginning of this section face the risk that a particular pattern occurring in nature (for example, an article about the North Korean Olympic team that combines traits of the "foreign policy" and "sports" articles) will not occur in the data set. Techniques like *Laplace smoothing*, which assigns probability $1/(n+k)$ to an unobserved outcome of an event with k possible outcomes and n observations, accommodate these exceptional cases. Many of the crash events recorded by CER lacked several fields, so we also faced the challenge of defining default values. This is actually easier to do for non-numeric data types: for strings and hexadecimal codes, our default value was defined to be different from all other instances of that data type. For set-valued data, the default value was the empty set; most of the applications for which we had no dependency information were statically-linked research applications, so they had no runtime dependencies. For numerical fields, we used the average over all observations of that field as an unbiased placeholder.

Clustering Analysis Results

In the absence of cluster exemplars, we are left with the task of manually inspecting the clusters found by our application. Figure 25 shows examples of clusters obtained from our data. Both agglomerative and *k*-means clustering successfully identified crash chains as collections of related events. Crash events in a chain shared almost all features in common, differing only in their time stamps, so these would comprise the most prominent groups in our very sparse data set.

Working our way from the leaves to the root of the cluster tree generated by the agglomerative algorithm, the clusters become harder to interpret. At the level of 25 clusters, the output of the agglomerative algorithm[1] includes a mix of stand-alone crash chains, user-specific crash histories (multiple crash chains attributed to the same user and

[1] We could not identify systematic differences between the results of the *k*-means and agglomerative algorithms, so we focus our presentation on the agglomerative results.

Microsoft Apps

Application	Component
iexplore.exe	mshtml.dll
explorer.exe	shimgvw.dll
iexplore.exe	ntdll.dll
explorer.exe	shell32.dll
mmc.Exe	comctl32.dll
OUTLOOK.EXE	FDATE.DLL
WINWORD.EXE	WINWORD.EXE
iexplore.exe	BROWSEUI.DLL
WINWORD.EXE	WINWORD.EXE
EXCEL.EXE	ntdll.dll
OUTLOOK.EXE	OUTLLIB.DLL
WINWORD.EXE	MSO.DLL
notepad.exe	comctl32.dll
POWERPNT.EXE	MSO.DLL
POWERPNT.EXE	ole32.dll
ntvdm.exe	ntdll.dll
ntvdm.exe	wow32.dll

Netscape

Application	Component
netscp.exe	gklayout.dll
netscp.exe	msgimap.dll
netscp.exe	gkplugin.dll
Netscp.exe	xpcom.dll

Mozilla Apps

Application	Component
thunderbird.exe	Hungapp
thunderbird.exe	gklayout.dll
thunderbird.exe	Unknown
thunderbird.exe	thunderbird.exe
firefox.exe	firefox.exe
firefox.exe	Hungapp
firefox.exe	kernel32.dll

Custom-written Apps

Application	Component
dialogeditor.exe	dialogeditor.exe
simpl_fox_gl.exe	simpl_fox_gl.exe
ADPS_ProjectBT3.exe	ADPS_ProjectBT3.exe
unison.win32-gtkui.exe	Hungapp
wfxctl32.exe	wfxut32l.dll
tphkmgr.exe	tphkmgr.exe
stratagus.exe	stratagus.exe
usrtogrp.exe	ntdll.dll
model_lr.exe	model_lr.exe
ray_tracing.exe	ray_tracing.exe
FlexPDE4.exe	FlexPDE4.exe
FlexPDE4.exe	Hungapp
allegro-ansi.exe	Hungapp
canvas5.exe	canvas5.exe

Figure 25: Sample application clusters. Each table shows a cluster of applications as decided by k-means and/or agglomerative clustering. The left column represents the application while the right column corresponds to a particular component used by the application.

workstation), and application-centric crash histories (crashes of the same application on different workstations). For example, one of the clusters contained seven crashes of sshclient.exe, including two on different computers with the same error code; another contained six crash chains generated by `netscape.exe` with a variety of offending DLLs (from the Netscape-specific `gklayout.dll` to the widely-used `msvcrt.dll`). Eight other clusters contained multiple crash chains experienced by a single user, but the offending DLLs of these crash chains rarely agreed. The oddest cluster we found contained only two applications, firefox.exe (the Mozilla Firefox browser) and alisp.exe (Allegro Common Lisp).

The largest clusters do not present unifying features. One cluster of the 25 contained 582 crash events. Most of these were user-terminated (reported as “hungapp”), but some identified DLLs as the cause of the failure. Many of these applications were published by a single large vendor, but smaller developers and open-source projects also made the list. Almost all of the cluster members, however, were “complex” interactive applications with large DLL sets.

Our analysis suggests that most crash histories are highly machine and/or user specific. Perhaps a high-level lesson we learned from this experience is that configuration management is a golden nugget in improving PC reliability. There is no single organization that is responsible for all crashes as the system instability is a result of incompatible configurations more often than it is due to bad application code. A useful tool in this direction would be an application compatibility checker that can verify upon application installation that it is safe to use given the current machine configuration.

Our analysis results seem promising that it is possible to find inter-application structural similarities given extensive crash data. However, to derive trustworthy patterns, we require orders of magnitude more crash data. It is important to have several instances of crashes generated for every application/component/error code tuple. Given the limited number of machines in the department and proficient system administration, it would be

difficult to generate the necessary data locally. We would derive more accurate clustering results if we ran the tool on a large scale of data such as the millions of crash reports collected by Microsoft.

It is very important to incorporate expert knowledge of these applications and their structure into our analysis engine. It is impossible to automatically capture all the design intricacies and functional descriptions that an informed application developer might readily provide. However, such immense domain knowledge may bias clustering results. So it is important to be aware of any bias introduced by expert knowledge while preserving the necessary structural information about each application. Additionally, it is also important to include end-user-experience in availability metrics that we use. Crash patterns do not always correlate with usage patterns and such information is instrumental in accurate analyses that normalize the data.

References

[And03] D. Anderson, "Public Computing: Reconnecting People to Science," *The Conference on Shared Knowledge and the Web*, Residencia de Estudiantes, Madrid, Spain, Nov. 2003.

[BS+02] P. Broadwell, N. Sastry and J. Traupman, "FIG: A Prototype Tool for Online Verification of Recovery Mechanisms," *Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, New York, NY, June 2002.

[BC+02] A. Brown, L. Chung, and D. Patterson, "Including the Human Factor in Dependability Benchmarks," *In Proc. 2002 DSN Workshop on Dependability Benchmarking*, Washington, D.C., June 2002.

[BS97] A. Brown and M. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," *In Proc. 1997 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Seattle, WA, June 1997.

[FM00] J. Forrester and B. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," *In Proc. 4th USENIX Windows System Symposium*, Seattle, WA, Aug. 2000.

[GW+04] A. Ganapathi, Y. Wang, N. Lao and J. Wen, "Why PCs are Fragile and What We Can Do About It: A Study of Windows Registry Problems," *In Proc. International Conference on Dependable Systems and Networks (DSN-2004)*, Florence, Italy, June 2004.

[GP05] A. Ganapathi and D. Patterson, "Crash Data Collection: A Windows Case Study," *To Appear in Proc. International Conference on Dependable Systems and Networks (DSN-2005)*, Yokohama, Japan, June 2005.

[Gra86] J. Gray, "Why Do Computers Stop and What Can Be Done About It?" *Symp on Reliability in Distributed Software and Database Systems*, pp 3–12, 1986.

[Gra90] J. Gray, "A census of Tandem system availability between 1985 and 1990," *Tandem Computers Technical Report 90.1*, 1990.

[GS04] J. Gray and A. Szalay, "Where the rubber meets the sky:bridging the gap between databases and science," Microsoft Research TR-2004-110, 2004.

[KK+04] A. Kalakech, K. Kanoun, Y. Crouzet and J. Arlat, "Benchmarking the dependability of Windows NT4, 2000 and XP," *In Proc. International Conference on Dependable Systems and Networks (DSN-2004)*, Florence, Italy, June 2004.

[Kal98] M. Kalyanakrishnam, "Analysis of Failures in Windows NT Systems," Masters Thesis, Technical report CRHC 98-08, University of Illinois at Urbana-Champaign, 1998.

[KK+99] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. "Failure data analysis of a LAN of Windows NT based computers," In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, 1999.

[KD+05] S. King, G. Dunlap and P. Chen, "Debugging operating systems with time-traveling virtual machines", *Proceedings of the 2005 Annual USENIX Technical Conference* , April 2005.

[KD00] P. Koopman and J. DeVale, "The Exception Handling Effectiveness of POSIX Operating Systems," *IEEE Trans. on Software Engineering*, Vol 26, No 9, pp 837-848 Sept. 2000.

[LR01] L. Lancaster and A. Rowe, "Measuring real-world data availability," In *Proceedings of LISA 2001*, 2001.

[LI95] I. Lee and R. Iyer, "Software Dependability in the Tandem GUARDIAN Operating System," *IEEE Trans. on Software Engineering*, Vol 21, No 5, pp 455-467, May 1995.

[Lev89] Y. Levendel, "Defects and Reliability Analysis of Large Software Systems: Field Experience," *Digest 19th Fault-Tolerant Computing Symposium*, pp 238-243, June 1989.

[MR+04] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, and M. Baker, "The lockss peer-to-peer digital preservation system," ACM Transactions on Computer Systems (TOCS), 2004.

[Mur04] B. Murphy, "Automating Software Failure Reporting," *ACM Queue* Vol 2, No 8, Nov. 2004.

[MG95] B. Murphy and T. Gent, "Measuring system and software reliability using an automated data collection process," *Quality and Reliability Engineering International*,Vol 11,1995.

[OB+02] D. Oppenheimer, A. Brown, J. Traupman, P. Broadwell, and D. Patterson, "Practical issues in dependability benchmarking," *Workshop on Evaluating and Architecting System dependabilitY (EASY '02)*, San Jose, CA, Oct. 2002.

[SS72] M. Schroeder, and J. Saltzer, "A Hardware Architecture for Implementing Protection Rings," *Communications of the ACM* Vol. 15, No. 3, pp. 157-170, March 1972.

[SK+00] C. Shelton, P. Koopman, K. DeVale, "Robustness Testing of the Microsoft Win32 API," *In Proc. International Conference on Dependable Systems and Networks (DSN-2000)*, New York, June 2000.

[SK+02] C. Simache, M. Kaaniche, A. Saidane, "Event log based dependability analysis of Windows NT and 2K systems," *In Proc. 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)***,** pp 311-315, Tsukuba, Japan, Dec. 2002.

[SC91] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability--a study of field failures in operating systems," In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, 1991.

[SM+04] M. Swift, Muthukaruppan, B. Bershad and H. Levy, "*Recovering Device Drivers*," in *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[TI92] D. Tang and R. Iyer, "Analysis of the VAX/VMS Error Logs in Multicomputer Environments – A Case Study of Software Dependability," *International Symposium on Software Reliability Engineering*, Research Triangle Park, North Carolina, Oct 1992.

[TI96] A. Thakur and R. Iyer, "Analyze-NOW-an environment for collection and analysis of failures in a network of workstations," *IEEE Transactions on Reliability*, R46 (4), 1996.

[TI+95] A. Thakur, R. Iyer, L. Young and I. Lee, "Analysis of Failures in the Tandem NonStop-UX Operating System," *International Symposium on Software Reliability Engineering*, Oct 1995.

[WL+93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient Software-Based Fault Isolation," In *Proc. Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, December 1993, pages 203 - 216.

[WC+01] M. Welsh, D. Culler and E. Brewer, "SEDA, an Architecture for well-conditioned scalable Internet Services," *18th Symposium on Operating System Principles*. Chateau Lake Louise, Canada, October 2001.

[WM+02] D. Wilson, B. Murphy and L. Spainhower, "Progress on Defining Standardized Classes for Comparing the Dependability of Computer Systems," *In Proc. DSN 2002 Workshop on Dependability Benchmarking*, Washington, D.C., June 2002.

[XK+99] J. Xu, Z. Kalbarczyk and R. Iyer, “Networked Windows NT system field failure data analysis,” In *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, 1999.