Understanding and Controlling Non-Determinism in Linux Services

by

Syed Aunn Hasan Raza

S.B., Computer Science and Engineering, M.I.T., May 2010

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author . . . .
Department of Electrical Engineering and Computer Science
June 21, 2011

Certified by . . . .
Dr. Saman P. Amarasinghe
Professor
Thesis Supervisor

Accepted by . . . .
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Understanding and Controlling Non-Determinism in Linux Services

by

Syed Aunn Hasan Raza

Submitted to the Department of Electrical Engineering and Computer Science
on June 21, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Both server and desktop virtualization rely on high VM density per physical host to reduce costs and improve consolidation. In the case of *boot-storms*, such high VM density per host can be a problem....

(To be filled in)

Thesis Supervisor: Dr. Saman P. Amarasinghe
Title: Professor

Acknowledgments

I would like to thank Professor Saman Amarasinghe for his huge role in both this project and my wonderful undergraduate experience at MIT. Saman was my professor for 6.005 (Spring 2008), 6.197/6.172 (Fall 2009) and 6.035 (Spring 2009). These three exciting semesters not only convinced me of his unparalleled genius, they also ignited my interest in computer systems. Over the past year, as I have experienced the highs and lows of research, I have really benefited from Saman's infinite insight, encouragement and patience.

As an M-Eng student, I have been blessed to work with two truly inpsirational and gifted people from the COMMIT group: Marek Olszewski and Qin Zhao. Marek and Qin's expertise and brilliance is probably only eclipsed by their humility and helpfulness. I have learned more from them than I probably realize, and their knowledge of Dynamic Instrumentation and Operating Systems is invaluable and, frankly, immensely intimidating. I hope to emulate (or even approximate) their excellence some day.

This past year, I have also had the opportunity to work with Professor Srini Devadas, Professor Fraans Kaashoek, and Professor Dina Katabi as a Teaching Assistant for 6.005 and 6.033. It has been an extraordinarily rewarding experience, and I have learned tremendously from simply interacting with these peerless individuals. Professor Dina Katabi was especially kind to me for letting me work in G916 over the past few months.

I would like to thank Abdul Munir, whom I have known since my first day at MIT; I simply don't deserve the unflinchingly loyal and supportive friend I have found in him. I am also indebted to Osama Badar, Usman Masood, Brian Joseph, and Nabeel Ahmed for their unrelenting support and encouragement; this past year would have been especially boring without the never-ending arguments and unproductive 'all-nighters' that accompany our friendship. I also owe a debt of gratitude to my partners-in-crime Prannay Budhraja, Ankit Gordhandas, Daniel Firestone and Maciej Pacula, who have been great friends and collaborators over the past few years.

I am humbled by the countless sacrifices made by my family in order for me to be where I am today. My father has been the single biggest inspiration and support in my life since childhood. He epitomizes, for me, the meaning of selflessness and resilience in life. This thesis, my work and few achievements were enabled by – and dedicated to – him, my mother and my two siblings Ali and Zahra. Ali has been a calming influence during my years at MIT; the strangest (and most unproductive) obsessions unite us, ranging from Chinese *Wuxia* fiction to, more recently, *The Game of Thrones*. Zahra's high-school problems have been a welcome distraction over the past year; they have also allowed me to appear smarter than I truly am.

Finally, I would like to thank my wife Amina for her unwavering love and support throughout my stay at MIT, for improving and enriching my life every single day since I have known her, and for knowing me better than even I know myself. Through her, I have also met two exceptional individuals, Drs. Fatima and Anwer Basha, whom I have already learnt a lot from.

> *"It is impossible to live without failing at something, unless you live so cautiously that you might as well not have lived at all – in which case, you fail by default."*
>
> J.K. Rowling, Harvard Commencement Speech 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Data centers increasingly use server virtualization to reduce operating costs, simplify administrative tasks and improve performance scalability. Through virtualization, it is possible to achieve high resource utilization and isolation at the same time: each application is typically assigned a dedicated server virtual machine (VM), while many VMs are consolidated on powerful host computers to reduce wasted cycles. The use of techniques such as memory overcommitment (including *transparent page sharing, ballooning* and *hypervisor swapping*) [18] has further improved the consolidation ratios and cost-effectiveness of server virtualization, and augurs well for the future of the technology.

Given the success of server virtualization, many companies are extending the use of virtualization to their desktop computers. In a Virtual Desktop Infrastructure [17] (VDI), desktop operating systems and applications are hosted in virtual machines that reside in a data center; users access virtual desktops from desktop PCs or thin clients via a remote display protocol. A VDI provides simplicity in administration and management: applications can be centrally added, deleted, upgraded and patched. VDI deployments also promise even higher consolidation ratios than those achieved via server virtualization because desktop virtual machines typically require less resources than server virtual machines.

Consolidation ratios (measured by VM density per host) in data centers are expected to increase in the future, not only because of improvements in virtualization technology, but also because new generations of processors support more cores and more memory [4]. Because a single VM would typically utilize only a modest fraction of a host's hardware resources, a high VM density per host is desireable in most cases for effective resource utilization. However, correlated spikes in the CPU/memory usage of many VMs can suddenly cripple host machines. For instance, a *boot storm* [4, 6, 9, 14, 16] can occur after some software is installed or updated, requiring hundreds or thousands of identical VMs to reboot at the same time. Bootstorms can be particularly frequent in VDIs because users typically show up to work at roughly the same time in the morning each day.

Concurrently booting VMs create unusually high I/O traffic, generate numerous disk and memory allocation requests, and can saturate host CPUs. To avoid the prohibitively high boot latencies that result from boot storms, data centers usually either boot machines in a staggered fashion, or invest in specialized, expensive and/or extra-provisioned hardware for network/storage [5, 6]. There is also anecdotal evidence that VDI users sometimes leave their desktop computers running overnight to avoid morning boot storms; this practice represents an unnecessary addition to already exorbitant data center energy bills [13]. Data deduplication [3], through which hosts reclaim/reuse disk blocks common to several VMs, has been proven to reduce the memory footprint of concurrently booting machines. However, while data deduplication can mitigate the stress on the memory subsystem in a boot storm, lowered memory latency can in turn overwhelm the CPU, fibre channel, bus infrastructure or controller resources and simply turn them into bottlenecks instead [10].

With the spread of virtualization, it is important to address the bootstorm problem in a way that does not involve simply skirting around the issue. Data deduplication is partly effective because identical VMs load the same data from disk when they boot up. In this thesis, we pose the following question: is it possible to generalize deduplication of data to deduplication of *execution*? If many identical VMs are concurrently booting up in a data center, do they execute the same set of instructions?

Even if there are some differences in the instructions executed, are they caused by controllable sources of non-determinism? Ultimately, if there is a way to ensure that concurrently booting VMs execute mostly the same set of instructions and perform the same I/O requests, one way to solve the boot storm problem may be remarkable simple in essence: instead of booting N identical VMs concurrently, we can boot one VM as a leader; the remaining $(N-1)$ VMs minimally follow the leader by executing a tiny subset of the instructions they would otherwise execute; we fork execution into N different instances as late as possible into the boot process. This approach could potentially reduce pressure on the underlying host hardware, and thereby enable data centers to handle boot storms effectively.

1.2 Goal of Thesis

This thesis aims to address the following questions:

1. When identical VMs boot up concurrently, how similar are the sets of instructions executed? What is the statistical profile of any differences in the distinct instruction streams?

2. What are the source(s) of any differences in the instruction streams of concurrently booting VMs? Are there ways to minimize the non-determinism in booting VMs?

The answers to these questions clearly are crucial in determining the feasibility of *deduplication of execution* as a possible solution to the boot storm problem.

1.3 Contrbutions

For this work, we used dynamic instrumentation frameworks such as Pin [7] and DynamoRio [2] to study user-level instruction streams from a a few representative Linux services at boot-time.

Specifically, we:

1. show that nondeterminism in Linux services (such as `cron`, `cups` and `ntp`) is bursty and extremely rare;

2. document the sources of non-determinism in Linux services – both obvious and obscure – and specify strategies for overcoming them in the boot storm scenario;

3. use simple Dynamic Instrumentation techniques to show that *fully* deterministic execution is achievable without *any* modifications to Linux or an executing service.

Strategies to achieve deterministic execution have been studied at the operating system layer [1] before, but they require modifications to Linux. Deterministic execution can be achieved in multi-threaded programs using record-and-replay approaches [12] or deterministic logical clocks [11]. Our study of non-determinism has somewhat different goals from from both approaches: we wish to avoid changing existing software to ease adoption; we also make several distinct – and potentially semantically different – executions *overlap* as much as possible, rather than replay one execution over and over. In our case, we do not know *a priori* whether two executions will behave identically or not. That the behavior of system calls or signals in Linux can lead to different results or side-effects across multiple executions of an application is well known: what is not documented is the application *context* in which these sources of non-determinism originate. To the best of our knowledge, this is the first attempt to study the statistical profile and context of non-determinism in Linux services in such detail. While we we hope this work ultimately proves the basis for an implementation of our proposed solution to the boot storm problem, we note that deterministic execution can immediately improve the effectiveness of existing virtualization technologies such as transparent page sharing and data deduplication.

1.4 Importance of Deterministic Execution

While our study of nondeterminism is driven by a specfic application, deterministic execution of programs can be beneficial in many different scenarios in its own right. Our work complements existing work on deterministc execution because it focuses on deterministic execution of primarily single-threaded services in Linux, at the granularity of individual instructions and their side-effects in memory. The motivations for deterministic multhreading listed in [11, 12] apply to our work as well.

Mainstream Computing and Security: If repeated executions of the same program can be expected to execute mostly the same set of instructions, then any significant deviations can be used to detect security attacks. Detection of such anomalous executions is the focus of *mainstream computing* [15], and deterministic execution obviously helps in reducing false positives.

Testing: Deterministic execution in general facilitates testing, because outputs and internal state can be checked at certain points with respect to expected values. Our version of determinism allows for a particularly strong kind of test case that may be necessary for safety-critical systems: with deterministic execution, a program must execute the exact same instructions across different executions, for the same inputs.

Debugging: Erroneous behavior can be more easily reproduced via determininstic execution, which helps with debugging. Deterministic execution has much lower storage overhead than traditional record-and-replay approaches.

1.5 Thesis Organization

In what follows, Chapter 2 presents an overview of the Linux boot process and the dynamic instrumentation techniques we used to profile non-determinism in Linux services. Chapter 3 presents a summary of the sources of non-determinism we encountered in this work and how we overcame them. Chapter 4 presents a detailed case study of a few selected Linux services to identify the context in which non-determinism arises, and the strategies that can be used to control it. Chapter 5 presents design ideas for an implementation of deduplication of execution. Chapter 6 summarizes related work. Finally, Chapter 7 concludes this thesis and discusses future work.

Chapter 2

Execution Profile of Linux Services

This chapter first provides some background on the Linux startup process (Section 2.1). It then describes how we collected user-level instruction streams from some Linux services via dynamic instrumentation to measure nondeterminism in the linux boot process (Section 2.2); finally, it summarizes our results on the statistical nature of nondeterminism in Linux services (Section 2.3).

2.1 The Linux Boot Process

When a computer boots up:

1. The BIOS (Basic Input/Output System) gets control and performs startup tasks for the specific hardware platform.

2. Next, the BIOS reads and executes code from a designated boot device that contains part of a Linux boot loader. Typically, this smaller part (or phase 1) loads the bulk of the boot loader code (phase 2).

3. The boot loader may present the user with options for which operating system to load (if there are multiple available options). In any case, the boot loader loads and decompresses the operating system into memory; it sets up system hardware and memory paging; finally, it transfers control to the kernel's `start_kernel()` function.

4. The `start_kernel()` function performs the majority of system setup (including interrupts, remaining memory management, device initialization) before spawning the `idle` process, the scheduler and the user-space `init` process.

5. The scheduler effectively takes control of system management, and kernel stays idle unless externally called from now on.

6. The `init` process executes scripts that set up all non-operating system services and structures in order to allow a user environment to be created, and then presents the user with a login screen.

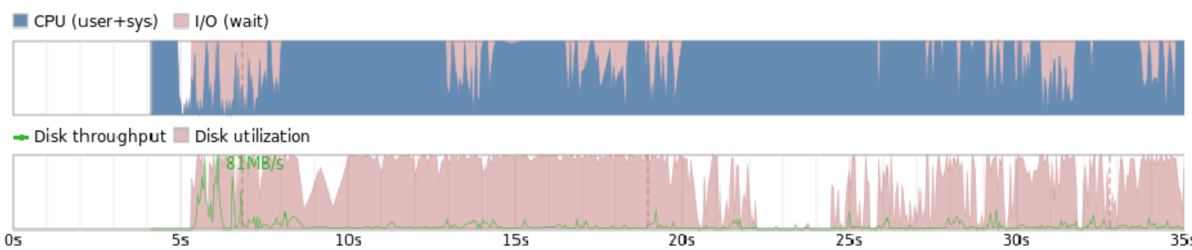


Figure 2-1: CPU and disk activity for a booting Ubuntu VM (0-35 seconds after `init` is spawned). The first few seconds show no activity because the data collection daemon takes a few seconds to start.

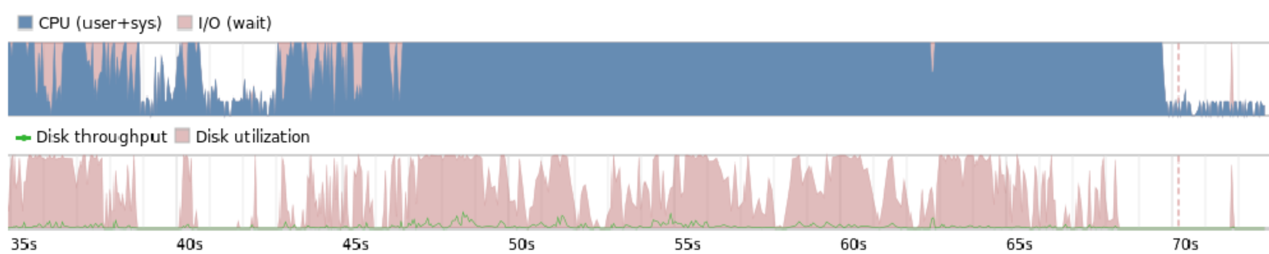


Figure 2-2: CPU and disk activity for a booting Ubuntu VM (35-73 seconds after `init` is spawned).

Figures 2-1 and 2-2 illustrate the CPU usage and disk activity of an Ubuntu 10.10 VM that takes about 70 seconds to complete the sixth step of the boot process (spawning the `init` process to set up the user environment). The Linux kernel version is 2.6.35-27-generic and the VM is configured with a single core processor with 512 Mb RAM. Generated using the Bootchart utility [8], the figures illustrate that the booting process involves high memory and CPU overhead (5-70 seconds); they also show a glimpse of the well-known fact that memory and CPU overhead typically dimishes greatly after the boot process is completed and the machine is ready for login (70+ seconds). This disparity in CPU/memory usage is the source of the boot storm problem; a single host can handle many VMs in steady-state usage, but the host gets crippled when the same VMs boot up concurrently.

In the last step of the booting process (step 6), `init` typically runs many scripts located in specific directories such as '`/etc/rc`' or '`/etc/init.d/`'. While the myriad Linux distributions can have their own variants of `init` binaries (e.g. `SysV`, or `systemd` or `Upstart`), the `init` process always directly/indirectly launches several services and daemons to initialize the user desktop environment. Figure 2-3 provides a summary of the specific actions performed by `init` (through the subprocesses or daemons it launches) for the same Ubuntu VM used for Figures 2-1 and 2-2.

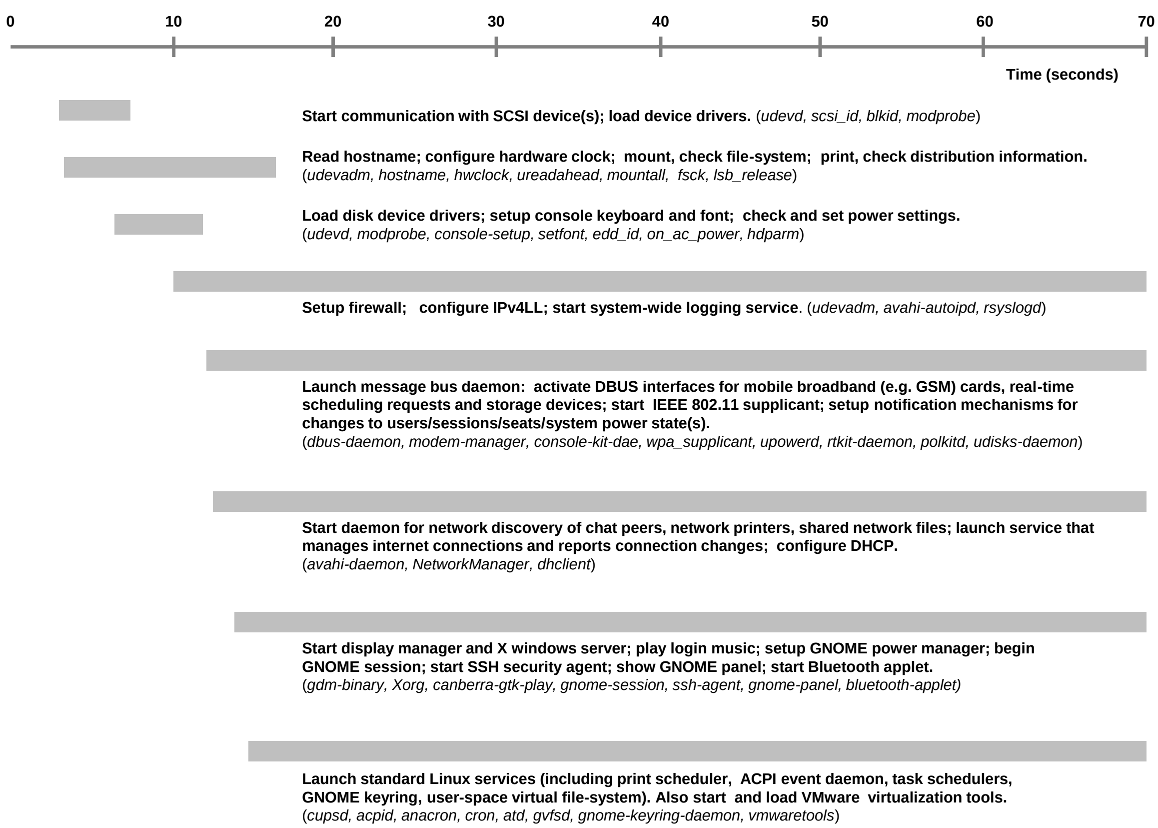


Figure 2-3: A summary of the actions performed by `init` for a booting VM; this figure has the same timeline (0-70 seconds) as Figures 2-1 and 2-2.

In fact, the `init` process actually launched 361 children processes (directly and indirectly) over the 70 second period summarized by Figure 2-3. Most of them were ephemeral processes; several processes were repeatedly launched in different contexts (e.g. `getty` or `grep`). The processes singled out in Figure 2-3 are the ones that either stayed alive through most of the boot process till the end, performed important boot actions, or spawned many sub-processes themselves.

2.2 Data Collection Scheme

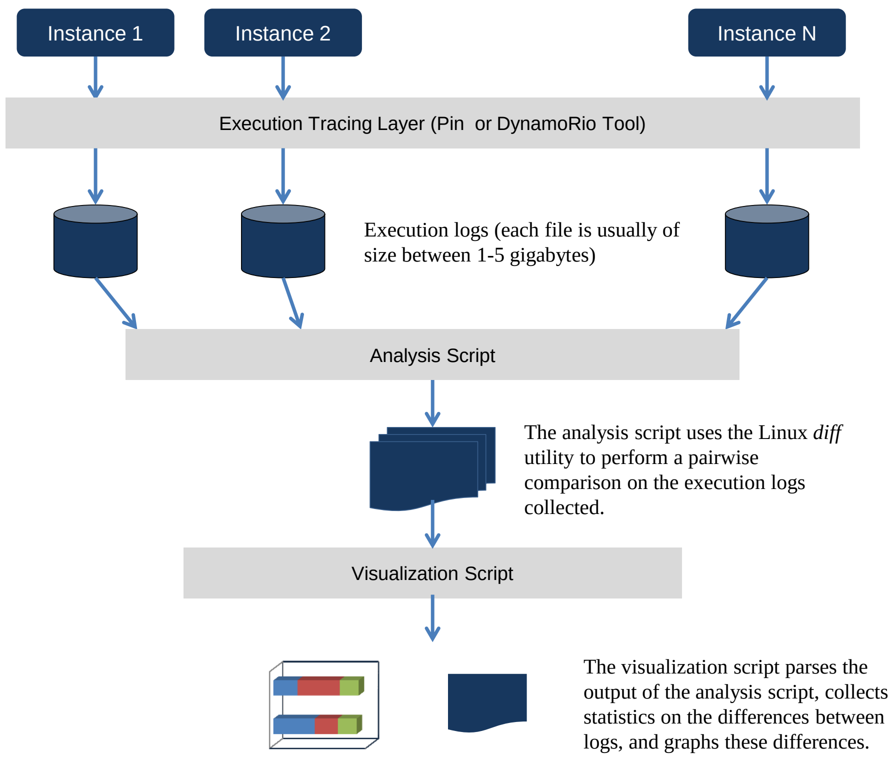


Figure 2-4: Steps involved in measuring execution nondeterminism.

Pin and DynamoRio are runtime frameworks that enable inspection and arbitrary transformation of user-mode application code as it executes. Using both Pin and DynamoRio to study the behavior of Linux services together allowed us to verify the the accuracy of our results. We used Pin more than DynamoRio because it gets injected into application code earlier than DynamoRio, which allows for greater instruction coverage for our purpose.

Figure 2-4 shows the simple steps involved in collecting data on nondeterminism using dynamic instrumentation. The next section explains each of these steps in detail, using a simple “Hello, world!” program as an example.

2.2.1 Analyzing a simple “Hello, world!” program

This section outlines the data collection scheme described in Figure 2-4 in detail with the help of an example: the simple “Hello, world!” program outlined in Figure 2-5. For this example, we disabled ASLR (Address Space Layout Randomization) on the Ubuntu VM described in section 2.1.

```c
#include <stdio.h>

int
main(int argc, char* argv[])
{
  printf(``Hello, world!\n'');
  return 0;
}
```

Figure 2-5: A “Hello, world!” program in C.

Execution Tracing Layer

As shown in Figure 2-4, the first step in data collection involves running the target program a few times across identical VMs. Ideally, these different executions are done concurrently. In our scheme, we wrote a Pin tool that:

1. logs each x86 instruction executed by the target process, along with the new values of any affected registers,

2. records values written to or read from memory,

3. intercepts all signals received, and records the instruction counts corresponding to the timing of any signals, and

4. monitors all system calls made by the target process, and logs any corresponding side-effects to memory or registers.

Implementation of the execution tracing layer required a detailed understanding and close examination of the Linux system call interface. Figure 2-6 shows an excerpt from a trace file generated by our Pin tool while running the “Hello, World” program. Our tool records and analyzes every instruction executed in user-space by the process for the “Hello, world” program once Pin gets control; this allows us to include program initialization and library code in our analysis.

```
_dl_make_stack_executable       0xb7ff6210  pop edx                              $ edx = 0xb7ff1040, esp = 0xbffff29c
        Read 0xbffff2d4 = *(UINT32*)0xbffff29c
_dl_make_stack_executable       0xb7ff6211  mov ecx, dword ptr [esp]             $ ecx = 0xbffff2d4
_dl_make_stack_executable       0xb7ff6214  mov dword ptr [esp], eax
        Write *(UINT32*)0xbffff29c = 0xb6415b90
        Read 0xb7fff8f8 = *(UINT32*)0xbffff2a0
_dl_make_stack_executable       0xb7ff6217  mov eax, dword ptr [esp+0x4]         $ eax = 0xb7fff8f8
        Read 0xb6415b90 = *(UINT32*)0xbffff29c
_dl_make_stack_executable       0xb7ff621b  ret 0xc                              $ esp = 0xbffff2ac
__libc_start_main       0xb6415b90  push ebp                                     $ esp = 0xbffff2a8
        Write *(UINT32*)0xbffff2a8 = 0
__libc_start_main       0xb6415b91  mov ebp, esp                                 $ ebp = 0xbffff2a8
__libc_start_main       0xb6415b93  push edi                                     $ esp = 0xbffff2a4
        Write *(UINT32*)0xbffff2a4 = 0x80482f0
__libc_start_main       0xb6415b94  push esi                                     $ esp = 0xbffff2a0
        Write *(UINT32*)0xbffff2a0 = 0x1
__libc_start_main       0xb6415b95  push ebx                                     $ esp = 0xbffff29c
        Write *(UINT32*)0xbffff29c = 0xb7ffeff4
__libc_start_main       0xb6415b96  call 0xb6415aaf                              $ esp = 0xbffff298
        Write *(UINT32*)0xbffff298 = 0xb6415b9b
        Read 0xb6415b9b = *(UINT32*)0xbffff298


 __NR_mmap2() called.
        addr = 0
        length = 4096
        prot = 3
        flags = 34
        fd = -1
        pgoffset = 0
        ret_val = b62dd000
 __NR_mmap2() returning.

 __NR_write() called.
        fd = 1
        pBuf = 0xb62dd000
        count = 14
        bytes written = 14
        buf contents:
        buf[0] = H
        buf[1] = e
        buf[2] = l
        buf[3] = l
        buf[4] = o
        buf[5] = ,
        buf[6] =
        buf[7] = w
        buf[8] = o
        buf[9] = r
        buf[10] = l
        buf[11] = d
        buf[12] = !
        buf[13] =
 __NR_write() returning.
```

Figure 2-6: An excerpt from the log files generated by the execution tracing layer. The top half shows the set of x86 instructions executed in user-space by the “Hello, world!” process, including instruction addresses, symbolic information (wherever available), affected register values and memory addresses. The lower half shows an excerpt from the system call log.

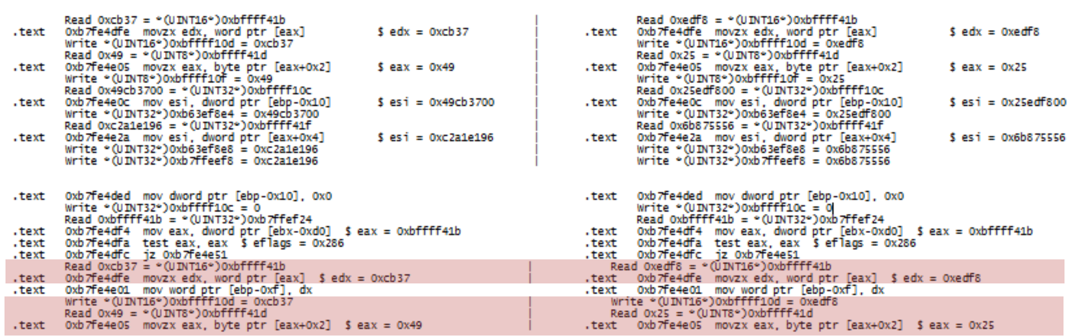
Analysis Script

The analysis script uses the Linux *diff* utility to perform pairwise comparisons of the log files generated by multiple executions of the target application. Using the `suppress-common`, `side-by-side` and `minimal` flags, the analysis script produces two output files:

1. A *delta* file that contains only instructions that were either conflicting between the two logs or missing in one log, and

2. A *union* file that contains all instructions executed in the two logs, but distinguishes instructions included in the delta file from matching instructions.

Figure 2-7 shows an excerpt from the union and delta files generated for the "Hello, world!" program. These generated files can be used to detect and diagnose sources of nondeterminism in an application. The delta and union files can be constructed from the two executions that are the most or least different, or have the median difference from the set of collected traces.



```
        Read 0xcb37 = *(UINT16*)0xbffff41b                                 |          Read 0xedf8 = *(UINT16*)0xbffff41b
.text   0xb7fe4dfe  movzx edx, word ptr [eax]         $ edx = 0xcb37       |  .text   0xb7fe4dfe  movzx edx, word ptr [eax]         $ edx = 0xedf8
        Write *(UINT16*)0xbffff10d = 0xcb37                                |          Write *(UINT16*)0xbffff10d = 0xedf8
        Read 0x49 = *(UINT8*)0xbffff41d                                    |          Read 0x25 = *(UINT8*)0xbffff41d
.text   0xb7fe4e05  movzx eax, byte ptr [eax+0x2]     $ eax = 0x49         |  .text   0xb7fe4e05  movzx eax, byte ptr [eax+0x2]     $ eax = 0x25
        Write *(UINT8*)0xbffff10f = 0x49                                   |          Write *(UINT8*)0xbffff10f = 0x25
        Read 0x49cb3700 = *(UINT32*)0xbffff10c                             |          Read 0x25edf800 = *(UINT32*)0xbffff10c
.text   0xb7fe4e0c  mov esi, dword ptr [ebp-0x10]     $ esi = 0x49cb3700   |  .text   0xb7fe4e0c  mov esi, dword ptr [ebp-0x10]     $ esi = 0x25edf800
        Write *(UINT32*)0xb63ef8e4 = 0x49cb3700                            |          Write *(UINT32*)0xb63ef8e4 = 0x25edf800
        Read 0xc2a1e196 = *(UINT32*)0xbffff41f                             |          Read 0x6b875556 = *(UINT32*)0xbffff41f
.text   0xb7fe4e2a  mov esi, dword ptr [eax+0x4]      $ esi = 0xc2a1e196   |  .text   0xb7fe4e2a  mov esi, dword ptr [eax+0x4]      $ esi = 0x6b875556
        Write *(UINT32*)0xb63ef8e8 = 0xc2a1e196                            |          Write *(UINT32*)0xb63ef8e8 = 0x6b875556
        Write *(UINT32*)0xb7ffeef8 = 0xc2a1e196                            |          Write *(UINT32*)0xb7ffeef8 = 0x6b875556

.text   0xb7fe4ded  mov dword ptr [ebp-0x10], 0x0                             .text   0xb7fe4ded  mov dword ptr [ebp-0x10], 0x0
        Write *(UINT32*)0xbffff10c = 0                                                Write *(UINT32*)0xbffff10c = 0
        Read 0xbffff41b = *(UINT32*)0xb7ffef24                                        Read 0xbffff41b = *(UINT32*)0xb7ffef24
.text   0xb7fe4df4  mov eax, dword ptr [ebx-0xd0]  $ eax = 0xbffff41b         .text   0xb7fe4df4  mov eax, dword ptr [ebx-0xd0]  $ eax = 0xbffff41b
.text   0xb7fe4dfa  test eax, eax  $ eflags = 0x286                           .text   0xb7fe4dfa  test eax, eax  $ eflags = 0x286
.text   0xb7fe4dfc  jz 0xb7fe4e51                                             .text   0xb7fe4dfc  jz 0xb7fe4e51
        Read 0xcb37 = *(UINT16*)0xbffff41b                                 |      Read 0xedf8 = *(UINT16*)0xbffff41b
.text   0xb7fe4dfe  movzx edx, word ptr [eax]  $ edx = 0xcb37              |  .text   0xb7fe4dfe  movzx edx, word ptr [eax]  $ edx = 0xedf8
.text   0xb7fe4e01  mov word ptr [ebp-0xf], dx                                .text   0xb7fe4e01  mov word ptr [ebp-0xf], dx
        Write *(UINT16*)0xbffff10d = 0xcb37                                |      Write *(UINT16*)0xbffff10d = 0xedf8
        Read 0x49 = *(UINT8*)0xbffff41d                                    |      Read 0x25 = *(UINT8*)0xbffff41d
.text   0xb7fe4e05  movzx eax, byte ptr [eax+0x2]  $ eax = 0x49            |  .text   0xb7fe4e05  movzx eax, byte ptr [eax+0x2]  $ eax = 0x25
```

Figure 2-7: Excerpts from the side-by-side diff files generated by the analysis script. The top half shows a few instructions at the start of the delta file; these instructions are different in the two logs (as indicated by the | in the middle of the line). The bottom half shows the corresponding instructions in the union file. Conflicting instructions are marked with the color red in the union file (along with the | symbol); the other instructions are found in both logs.

Visualization Script

The visualization script reads the "union" file to derive statistics on the extent of differences in the original logs, and generates a diagram to capture the different execution traces of the program.

In particular, it derives three key metrics from the "union" file:

1. *Length of Common Prefix* **(P):** This is the number of instructions common to both logs starting from the beginning and up to the point of first divergence.

2. *Longest Common Substring* **(LS):** This is the largest sequence of adjacent instructions that are common to both logs.

3. *Longest Common Subsequence* **(LCS):** Intuitively, this is the "overlap" in the logs; it is the length of the longest sequence of instructions found in both logs. Instructions in the LCS must be in the same order in both logs, but they are not required to be adjacent.

For instance, if the first instance of a program executes the instruction sequence $I_1 = [A, B, C, D, E, F]$, and the second instance of the same program executes the instruction sequence $I_2 = [A, B, X, D, E, F, Y]$, then: the common prefix is $[A, B]$; the longest common substring is $[D, E, F]$, and the longest common subsequence is $[A, B, D, E, F]$.

In general, the longest common subsequence (LCS) of the two traces is arguably the most indicative of the extent of determinism in two executions of a program. The other two metrics are important for evaluating the feasibility of deduplication of execution as a solution to the boot storm problem. In general, we want the common prefix (P) and the longest common substring (LS) of the two logs to be as large as possible to ensure that concurrently booting VMs do not need to branch execution or communicate with each other too quickly. This is further discussed in chapter 6.

For the "Hello, world!" program, if ASLR is enabled, the two logs have very little overlap ($< 5\%$), and the common prefix and longest common substring are on the order of 10 instructions. With ASLR disabled, one may expect the two traces

Table 2.1: Execution Statistics of "Hello, world!" program

Common Prefix	21.49 percent
Longest Common Substring	67.70 percent
Longest Common Subsequence	99.98 percent
Conflicts	0.02 percent

to look identical (because of the simplicity of the program), but there is still some nondeterminism in the instruction sequences (see Table 2.1 and Figure 2-8).

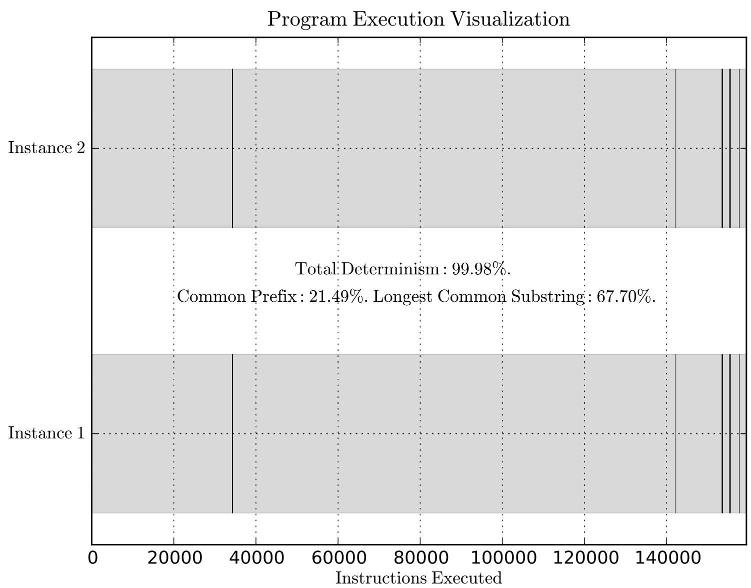


Figure 2-8: Visualization of "Hello, world!" program execution. The thin black lines represent conflicts between the two instances of the program.

2.3 Statistical Results

2.4 Summary

Chapter 3

Sources of Nondeterminism

This chapter summarizes the sources of non-determinism found in Linux services from a high-level; it also briefly describes the strategies we used to overcome these sources of non-determinism.

3.1 Linux Security Features

3.2 Randomization

3.3 Process Identification Layer

3.4 Time

3.5 File I/O

3.6 Network I/O

3.7 Signals

3.8 Inter-thread Communication/Scheduling

3.9 Misc System Calls

3.10 Summary

Chapter 4

Case Studies of Linux Services

This chapter summarizes the context and extent of non-determinism found in three Linux services (`cron`, `cupsd` and `ntp`) in detail.

4.1 Cups

4.2 Cron

4.3 Ntp

4.4 Summary

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.

Bibliography

[1] T. Bergan, N. Hunt, L. Ceze, and S.D. Gribble. Deterministic process groups in dos. *9th OSDI*, 2010.

[2] D.L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Citeseer, 2004.

[3] A.T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in san cluster file systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 8–8. USENIX Association, 2009.

[4] J.G. Hansen and E. Jul. Lithium: virtual machine storage for the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 15–26. ACM, 2010.

[5] Solving Boot Storms With High Performance NAS. `http://www.storage-switzerland.com/Articles/Entries/2011/1/3_Solving_Boot_Storms_With_High_Performance_NAS.html`, 2011. [Accessed 1-August-2011].

[6] X.F. Liao, H. Li, H. Jin, H.X. Hou, Y. Jiang, and H.K. Liu. Vmstore: Distributed storage system for multiple virtual machines. *SCIENCE CHINA Information Sciences*, 54(6):1104–1118, 2011.

[7] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[8] Bootchart: Boot Process Performance Visualization. `http://www.bootchart.org`, 2011. [Accessed 29-July-2011].

[9] S. Meng, L. Liu, and V. Soundararajan. Tide: achieving self-scaling in virtualized datacenter management middleware. In *Proceedings of the 11th International Middleware Conference Industrial track*, pages 17–22. ACM, 2010.

[10] VMware Bootstorm on NetApp. `http://ctistrategy.com/2009/11/01/vmware-boot-storm-netapp/`, 2009. [Accessed 29-July-2011].

[11] M. Olszewski, J. Ansel, and S. Amarasinghe. Scaling deterministic multithreading. *2nd WoDet*, 2011.

[12] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[13] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the electric bill for internet-scale systems. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 123–134. ACM, 2009.

[14] Vijayaraghavan Soundararajan and Jennifer M. Anderson. The impact of management operations on the virtualized datacenter. *SIGARCH Comput. Archit. News*, 38:326–337, June 2010.

[15] M.W. Stephenson, R. Rangan, E. Yashchin, and E. Van Hensbergen. Statistically regulating program behavior via mainstream computing. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 238–247. ACM, 2010.

[16] S.B. Vaghani. Virtual machine file system. *ACM SIGOPS Operating Systems Review*, 44(4):57–70, 2010.

[17] VMware Virtual Desktop Infrastructure. `http://www.vmware.com/pdf/virtual_desktop_infrastructure_wp.pdf`, 2011. [Accessed 29-July-2011].

[18] C.A. Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.