Out-of-Band Detection of Boot-Sequence Termination Events

Naama Parush Dan Pelleg Muli Ben-Yehuda Paula Ta-Shma
IBM Haifa Research Lab
{naamap,dpelleg,muli,paula}@il.ibm.com

ABSTRACT

The popularization of both virtualization and CDP technologies mean that we can now watch disk accesses of systems from entities which are not controlled by the OS. This is a rich source of information about the system's inner workings. In this paper, we explore one way of mining the stream of data, to determine if the system had finished booting. Systems which we detect as failing to boot (or taking too long to boot) are flagged for further manual or automatic remediation. By performing this detection out-of-band, we gain a head start on any detection scheme that runs within the OS, and therefore must wait for the boot event to finish. Additionally, our scheme is agnostic to file-system layout and to kernel architecture.

We implemented our solution for the x86 architecture under two different virtualization platforms, and tested it on both Windows and Linux virtual machines. Under a variety of workloads and configurations, our detector managed to successfully identify the boot termination event, in most cases within 5 seconds of the event.

Categories and Subject Descriptors: D.4.m.

General Terms: Algorithms.

Keywords: boot detection, out-of-band, virtualization.

1. INTRODUCTION

However you define it, it seems that identifying the end of the boot sequence is an easy exercise given full access to the machine—a carefully-placed startup script should do the trick. But what if your access is limited? The premise of this paper is to pinpoint this particular timing information, with the constraint that the OS itself cannot be modified — furthermore, that its very identity may be unknown.

Why is this an interesting question? First and foremost, the answer proves useful in a variety of scenarios where the OS is virtualized and running under some sort of hypervisor [4, 5]. In some of them, computation is in fact provided as a service, where clients send sealed virtual machine images to be hosted and run by the vendor (e.g., Amazon EC2 [1]). The hosting service can assume nothing on the content of the images, nor is it (contractually) allowed to modify them. By solving the question we pose, the vendor can offer additional monitoring services to its clients, without requiring additional work from them.

ICAC'09, June 15–19, 2009, Barcelona, Spain.
ACM 978-1-60558-564-2/09/06.

Second, this is applicable where there are large numbers of machines that boot from storage devices that can be independently monitored. One such case is booting over the network, for example by a diskless machine. Another is a machine that is backed up by some CDP [3] mechanism, which creates a data stream of its disk accesses that is sent over the network.

If the boot-detection scheme is indeed oblivious to the OS, it immediately follows that it can support varied makes and models of operating systems. So a single piece of detection code can be used to monitor a whole range of machines, eliminating the need to develop and support code for each and every possible OS version. And like in the virtualization scenario, this capability can be retrofitted to large pools of installed machines. By localizing the change to a single control point (be it the storage device, the backup drivers, or even a passive network sniffer), such a change is made immensely more economical.

Technologically, our key insight is that the boot sequence is repetitive across instances. This is because it is nearly spontaneous and does not rely on external inputs. We propose a method based on the interception of the I/O accesses, either by an hypervisor, or by some other storage or network layer. This allows us to observe the boot sequences from the perspective of the disk. Essentially, we look for recurring patterns in this stream. This has additional benefits, like the ability to detect when new software has been installed (in a way which modifies the boot sequence), or when the system failed to boot correctly.

2. ARCHITECTURE

In order to achieve the goal of adapting to any system on the fly, we introduce a training stage in which the particulars of the VM's boot sequence are inspected. This is where we learn a sequence of disk blocks that characterizes its start-up routine. We call this the *reference set.* After this set has been established, detection may start. During detection, the data read from the running system is compared to the reference set. If the live data meets a matching criterion, we declare that the boot sequence has terminated.

Training Process. We assume that different phases of the boot procedure can be characterized by the homogeneity of block numbers between different boot runs. For example, first the kernel image will be read, in sequential order. Afterwards, the start-up procedure may commence, with some variability in the block numbers. Finally, once the system is running, the variability will be highest. Therefore, we divide the block sequence into phases with differing levels of

homogeneity. When the block variation between different sequences increases significantly, we declare the boot sequence terminated. Below, we elaborate on the technical details of the process.

Dividing the training sequences into phases and determining the time of boot termination. We want our method to be agnostic to file-system layout. It immediately follows that the analysis is done at the block level. Therefore we base our metric, described below, on just the recurrences of the block *indices*. In other words, if some random permutation were applied to the disk indices, the metric would not change. Another guiding principle is that after boot, the system is in a steady state, and each block is accessed with some respective probability that is approximately constant over time. On the other hand, during boot, which is a singular (albeit long) event, the variation, per block, of access events, will be high. A well-known measure which conveniently captures variability is the Shannon (or information) entropy. Given a random variable X, the entropy $H(x)$ is defined as $-\sum_x P(x) \cdot \log_2(P(x))$. A low entropy value signifies uniformity, and vice versa. Therefore this metric is suitable for our purpose, as detailed below.

To implement our scheme, we partition all training sequences into *windows*. A window is a consecutive part of the input stream, which has a fixed number of data points (i.e., block numbers) in it. In each window, and for every block number, we calculate the entropy of the counts of block occurrence across all sequences. For example, assume the window size is six and that we have observed three training sequences. For a given window in the first sequence the block numbers, in order, are: {A,B,A,A,B,B}. For the same window in the second and third sequences they are {A,B,B,A,B,B} and {A,B,A,B,A,B}, respectively. Block A appears three times in the first sequence, twice in the second sequence, the three times in the third. Therefore the count vector for block A is $\langle 3, 2, 3\rangle$. and the corresponding entropy is 1.56 (this is the entropy of the vector $\langle 3/8, 2/8, 3/8\rangle$). Similarly, we compute the entropy for block B. Then we take the average over all block indices (two in this example).

3. EXPERIMENTS

We began by testing the boot detection algorithm on block sequences gathered from a Linux (SLES-10) machine. Immediately after booting, the VM ran one of the following workloads: iozone, libMicro, pChase[6], iperf, and *webload* — a local web client rapidly accessing a collection of static pages (the Apache documentation set) stored on a local Apache server. A boot detector was trained and tested on sequences from all the different loads, shuffled randomly.

We then repeated the tests on a Linux-based WebSphere (WAS) virtual appliance under KVM. Here, the system had three disks attached (for system, data, and applications). We performed tests on data from just the system disk, as well from all three disks. A third set of tests added background noise, in the form of another guest on the same physical host, running Microsoft Windows XP.

Lastly we tested Microsoft Windows XP under KVM. The workloads included: idle, a CPU-intensive benchmark [2], optionally preceded by a boot-time disk scan, and an idle machine with a simultaneous VM on the same host, running Linux and compiling a kernel source tree.

As shown in Figure 1, our algorithm successfully detected boot termination in all sequences. The maximum error is under 15 seconds, with the vast majority of the Linux-based results measuring under 5 seconds of error. For Microsoft Windows, the error is larger — most samples are within 10 seconds. We attribute much of that to the inherent inaccuracy of the ground-truth signal on this platform.

4. CONCLUSION

We presented an effective and generic method for out-of-band detection of boot sequence termination. Our method works by inspecting the data stream between a (virtual or physical) machine and its (virtual or networked) storage. It detects the point in time at which a machine finished performing the routine start-up tasks, and signals this event. Additionally, we can detect divergence from the regular sequence, either in addition or in place of the normal activity.

5. REFERENCES

[1] The Amazon Elastic Compute Cloud (Amazon EC2) web site. `http://aws.amazon.com/ec2`.

[2] BYTEmark. `http://www.byte.com/bmark/bdoc.htm`.

[3] FilesX. `http://www.filesx.com/`.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[5] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *OLS '07: Ottawa Linux Symposium*, pages 225–230, July 2007.

[6] D. Pase. The pChase benchmark page. `http://www.pchase.org/`.

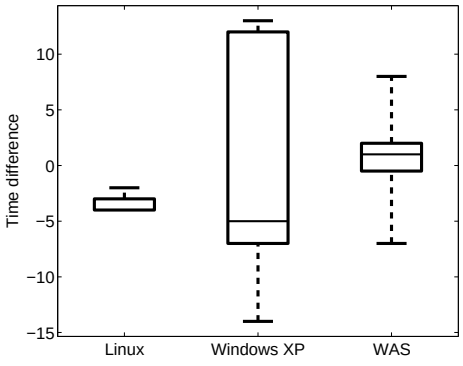

Figure 1: Summary of detection accuracy for various workloads. The box edges show the first and third quartile, and the middle line shows the median. The "whiskers" show the range of the data. Sample sizes are 156 for Linux, 146 for Windows XP, and 104 for WAS.