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

Naama Parush naamap@il.ibm.com
Dan Pelleg dpelleg@il.ibm.com
Muli Ben-Yehuda muli@il.ibm.com
Paula Ta-Shma paula@il.ibm.com

IBM Haifa Research Lab

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. This opens up the possibility of monitoring large pools of existing machines, with no need to modify their software or even notify their owners. We show that apart from the signaling of readiness for activity, we can potentially also detect major changes in the file layout of the system, which is a possible indication of intentional upgrades or malicious activity.

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.

1. INTRODUCTION

What does it mean for an operating system to have finished booting? Is it the point in time where it has finished powering up? Finished loading the kernel? Finished its initialization scripts? Started running services? Indeed, should the user even care, assuming she can get her work done? 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 [6, 8]. 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. It might also benefit the vendor itself, if it extracts important statistics on the inner workings of the hosted machines, and uses them to optimize its own operations.

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. 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 and level. 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 CDP drivers, or even a passive network sniffer), such a change is made immensely more economical.

Third, an OS image may have been audited for compliance with applicable regulations (the financial and healthcare sectors make good examples). Making even the slightest change to these images might trigger another costly certification cycle. Our approach eliminates the the need for such changes.

Finally, in some cases it may be desirable to track the system's progress even *before* it has completed initialization. This cannot be done with a simple flag-raising script. Our approach — as described below — can provide a "progress bar" that tracks the boot sequence at a fine level of detail. Below we show how this kind of close monitoring can result in detection of the presence of major changes to the file layout of the system, possibly due to a major upgrade or

perhaps a malicious piece of software.

This paper is about the technology needed for the detection of boot-sequence termination. Remediation and diagnosis is outside its focus. However, we can enumerate several operational scenarios where this capability would be useful. For example:

- Monitoring unattended servers (virtual or not) for correct operation. A failure to boot may, depending on the operational policy, trigger a restart, or alert a human operator.
- Creating a checkpoint that captures a system ready-to-run. This could be later used for (failure) recovery and fast-boot purposes.
- If there are dependencies among hosts, one may wish to delay starting some of them, according to some known topological order. For example, in a whole-site disaster-recovery scenario, an important DHCP or file server will need to be up before other machines are powered on. With our method, this kind of staged start-up could be automated.
- Testing of upgrades, patches, and configuration changes, especially ones that may cause a boot failure. For example, kernel or boot-loader modifications.

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[1]. 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.

In Section 2 we present an algorithm, based on statistical tests, to find the recurring patterns. We evaluate its performance in Section 3, discuss related work in Section 4, and conclude in Section 5.

2. ARCHITECTURE AND IMPLEMENTATION

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 are inspected. This is where we learn a sequence of disk blocks that characterizes its start-up routine[2]. 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 the system operational, for example, implying that a checkpoint may be taken. Below, we first elaborate on the training process and then provide details on the detection process and its possible outcomes.

[1]Some non-repeating events do affect its operation, such as DHCP server timing and randomization effects, but in our scheme they only add minor noise.

[2]In practice, the sequence consists of the just the first respective block IDs from each continuous sequence read from the disk.

2.1 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, the blocks that read the kernel image will always be read in order, before any initialization scripts are executed. 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.

2.1.1 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 clarify what we mean by block indices, these are the logical locations of the units of storage information, as seen by the system under inspection (be it virtualized or not). For example, if this is a virtual machine, and its storage is backed by a file on the file system of the host machine, then moving this file around the disk of the host, or relocating it to a different disk or to remote storage, would *not change* the block indices. In addition, we found it sufficient to ignore the writes, and only consider the read accesses. Therefore consider all the algorithms and experiments below to be ignoring writes (even though they could work just as well otherwise).

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. It can be thought of as measuring system time. For example, "seconds 5 through 10 since system was started". However, it does not exist on the time axis, but rather on the input stream axis. Thus it may contain the fifth through the tenth data items, in order of arrival.

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).

The window size (number of blocks per window) is a given parameter and our default value is the mean number of blocks per second plus 1.5 standard deviations. Figure 1 illustrates the mean entropy per window for data collected from 10 boot runs of a Linux VM. We see several phase transitions, where the most dramatic decrease at window 16 is defined as the time of boot termination.

We cluster the mean entropy to determine the phase transitions. The break points are initially chosen randomly and repeatedly improved by setting each partition break point independently while "freezing" the others.

We perform independent clusterings for each number of clusters in the range 2–5, and pick the partition with the minimum intra-cluster, and maximum inter-cluster, standard deviation. Once the best partition is set, the cluster representing the latest time period is removed (i.e., it encompasses the post-boot data points), and the remaining ones determine the phases as described above. In practice, we found results are best when the second and onwards phases are merged into a single phase (so there are at most two boot phases).

2.1.2 *Learning the reference sequence according to the phase transitions*

The reference sequence consists of blocks that appeared in at least a given percentage of the training sequences. The parameter indicating the needed rate of sequences is by default set to 0.8. In other words, if at least 80% of the training sequences include a specific block, then the block will be included in the reference set. The reference set is learned for each phase separately.

2.1.3 *Learning the matching criteria for the reference sequence*

To match a sequence to a reference set, we compare each of their respective phases separately. For each phase, we compute the *matching ratio*: the size of intersection between the live sequence and the reference set, relative to the size of the reference set. If this ratio exceeds the matching criterion, then the live sequence is considered a match to the reference. The matching criterion is set after the training is completed. To compute it, we take the minimum over the matching ratios of all pairs that include a training sequence and a reference sequence. We then multiply the result by a given parameter, typically set to 0.98. The matching ratio is denoted by M_1 in Figure 2 below. A sequence matches only if all of its phases match. Note that according to this scheme, there is no significance to the order of block numbers within a phase.

2.2 Detection process and possible outcomes

A sequence of read block numbers is given to the detection process, and compared to the reference boot sequence.

The comparison process is done sequentially over all phases. The blocks of the given sequence are compared to the blocks of the reference phase until its matching criterion is reached and the subsequent phase is tested. The detailed algorithm (for a given phase) is shown in Figure 2. The possible outcomes of the algorithm are:

Successful boot termination The boot sequence matched the reference sequence.

Successful boot termination, long sequence The boot sequence matched the reference sequence. However, the "mismatch", the rate of blocks from the tested sequence not present in the reference sequence, exceeded a given threshold (by default set to 0.7). This value is denoted as M_2 in Figure 2. This case may indicate that the boot consisted of an additional process such as a disk scan. The user might consider retraining the boot detector if this outcome repeats itself.

Timeout failure The boot sequence exceeded a maximum boot duration threshold (for example, five minutes) before reaching the matching criteria.

Unexpected death failure The sequence ended before reaching the matching criteria. This likely indicates a crash.

Failure, boot sequence change The sequence did not reach the matching criteria, and the "mismatch" exceeded the mismatch threshold. This outcome is strong evidence that the boot procedure has changed and should invoke retraining of the boot detector. This could be the result of a major OS upgrade or a big change in the initialization scripts or their order.

2.3 Implementation

Our implementation is divided into an I/O interceptor, which records the block numbers of each I/O access, and a classification module, which processes this data. In an early prototype, the interceptor recorded the data to a file, and the learning and classification was done off-line with MATLAB [4]. In a production version, the classifier is implemented in Python in about 2500 lines of code. The communication between the interceptor and the classifier is over a socket. We implemented interceptors for both the Xen [6] and KVM [8] hypervisors. In both cases the interceptor patch was well under 100 lines of code.

2.4 Complexity and Timing

Let T be the number of training sequences and N the maximal sequence length. During the training period, the algorithm mostly calculates the mean window entropy. The window entropy is calculated over each unique block number in each window, i.e. there are at most N different entropy calculations. Each entropy calculation takes at most $O(T)$ time, therefore each training window takes $O(T \cdot N)$ calculations and the whole training stage takes $O(N \cdot T^2)$. After training is over, the reference set is created by scanning the block numbers in all training sequences. Hence, this part is bounded by $O(N \cdot T)$ calculations. During detection of a live sequence, the algorithm compares the reference blocks to the sequence blocks, i.e. $O(N)$ calculations. Figures 3 and 4 illustrate the run time of a typical Linux VM boot detection procedure, measured on a standard dual-core, 3.7 GHz Intel machine. We can see that detection overhead is very low—well under 100 milliseconds in most cases. Also, the detection time is nearly constant, regardless of the sequence length.

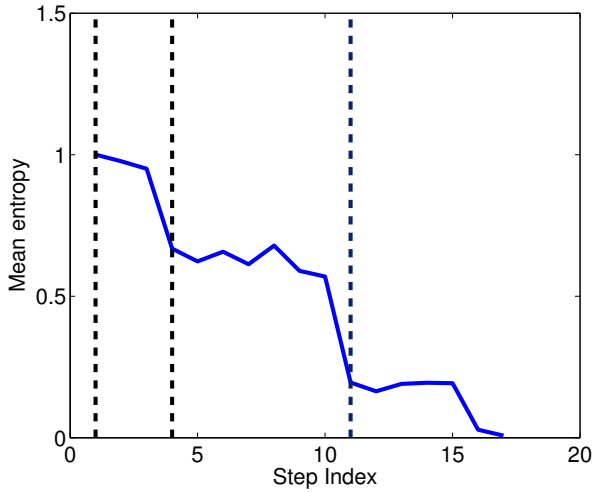

Figure 1: Window entropy over (system) time, as collected from 10 boot runs of a Linux VM.

3. EXPERIMENTS

Firstly, we tested 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[10] A popular I/O benchmark.

libMicro[15] A micro-benchmark on the memset call.

pChase[11] A CPU intensive benchmark.

iperf[16] A network benchmark run against the hypervisor as the remote node.

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. For calibration, we sampled the SSH port on the VM once a second. The first point in time at which the port was accepting connections is the one we treat as the ground truth for boot sequence termination. One may argue that this is a viable alternative to our approach. However, it requires knowledge that (a) SSH is installed on this VM; (b) SSH is a meaningful indicator of the VM's operation. Either of these breaks the black-box assumption.

Secondly, we 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.

Thirdly, we also tested a Microsoft Windows XP guest under KVM. The workloads included

- Idle.
- A CPU-intensive benchmark [2].
- A disk-scan during the boot sequence followed by the CPU benchmark.
- An idle machine. At the same time, on the same host, a different VM was running Linux which was compiling a kernel source tree.

For this guest, ground-truth calibration was done by a manually-installed startup script that fetched a (fictitious) web page from the monitoring host.

Our algorithm successfully detected boot termination in all sequences. See Figures 3 through 3 for detailed results, and Figure 8 for a summary. 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. Recall that in this case we rely on a user application (a web browser) starting up and performing uncontrolled operations such as DNS lookups and security-update checks before sending the control signal.

When we drill down into the data we find that the homogeneity of the training data can greatly skew the detection event timing. In early experiments, we used data from a single workload, repeated over multiple VM restarts, to both train the detector and then be used as live sequences. For example, in the Linux setup, for the "iozone" and "libMicro" workloads, the signaling of the termination event can be delayed considerably by up to 46 seconds (The normal boot sequence on this machine takes about 22 seconds.) At the same time, the signaling is not at random times, but rather highly concentrated within a range of about five seconds. Since the block sequences in these experiments were very similar even after the startup scripts finished running, the algorithm detected a very long sequence as part of the boot procedure. The same effect repeated in the WebSphere and Microsoft Windows experiments.

This behavior is consistent with our definition of the boot event as "some sequence that reliably follows a system startup event". It may be argued that this definition diverges from

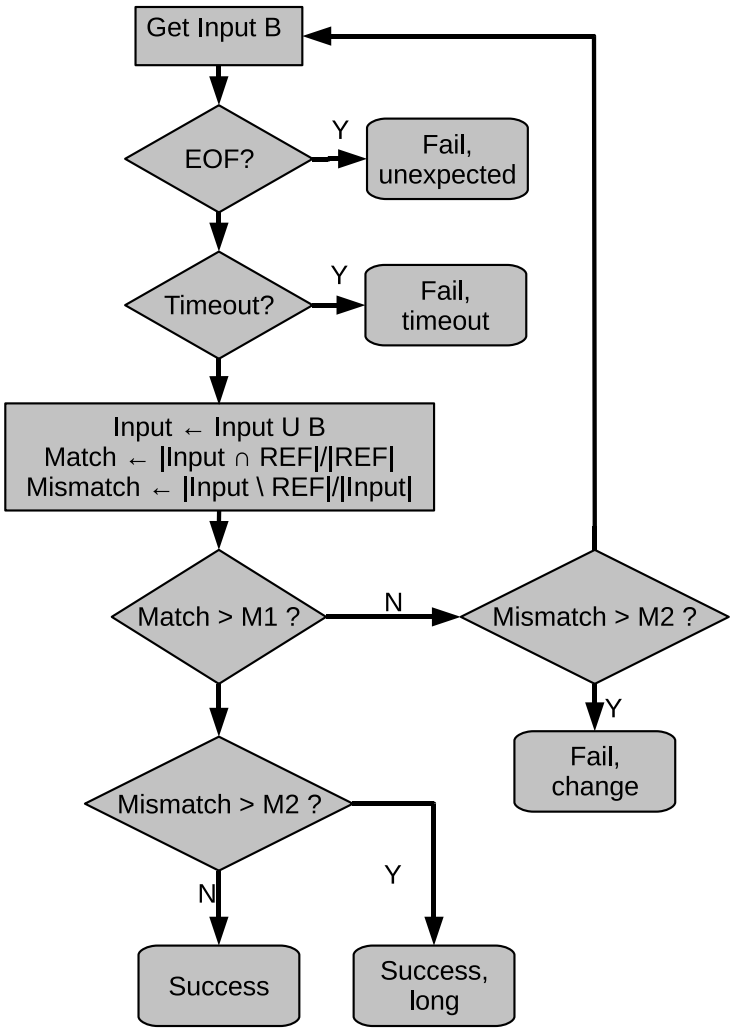

Figure 2: Flow chart of the boot detection algorithm. The thresholds M_1 and M_2 are described in the text.

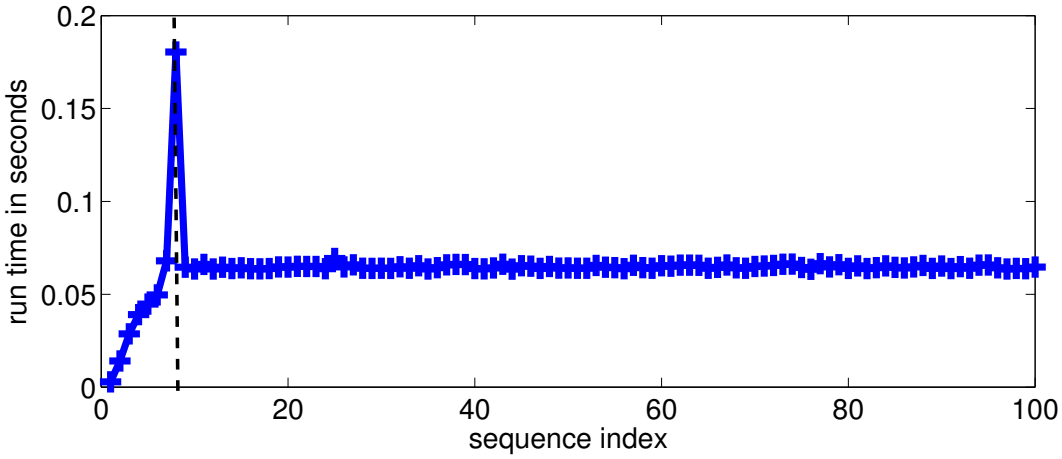

Figure 3: Run time, as a function of the sequence index. The marker line denotes the creation of the reference set (i.e., transition from training to learning.).

the point in time in which the OS finished setting up the stage and sits back while the applications take over. Which definition is best is an open question. In some scenarios, such as fast-boot and failure recovery, we feel our definition works best. In addition, we expect that in a real-life setup, there will be enough heterogeneity in the samples to avoid this phenomena. For example, several cloned VMs will probably be used to train a single classifier, and the plurality will generate the needed variability.

4. RELATED WORK

Access patterns to block storage have been studied in the past to improve the performance, reliability, and security of storage systems [5, 9]. This has typically been considered at the storage subsystem level and can therefore be considered as an out-of-band approach, similar to ours. Work on semantically-smart disk systems provides an out-of-band gray box approach where the disk subsystem uses prior knowledge of file system and database designs in order to reach conclusions about block level accesses [14, 13]. There are two drawbacks to this approach. Firstly, working solutions are difficult to maintain under file system or database format changes, which can happen relatively frequently [13]. Secondly, the approach is file and operating system specific. For example, the work on semantically-smart disk systems assumes an FFS-like file system layout [14]. In contrast, our approach treats the workload, encapsulated in a virtual machine, as a black box and uses machine learning to understand a particular semantic behavior of the virtual machine. We showed how this approach generalizes to operating and file systems with unrelated designs (specifically, Linux and Windows).

An out-of-band black box approach to mining block correlations is found in C-Miner [9]. These correlations can be used to improve storage subsystem performance, therefore the purpose is different from ours. Since the main application is pre-fetching, errors in prediction will at most result in performance degradation. In contrast, our detection is for a specific event and errors will therefore result in badly-timed checkpoints. This implies that the prediction rules in our application need to be far more strict in comparison.

Some continuous data protection products allow applications to mark time points suitable for recovery. This support is typically tied to specific application (e.g., a mail server). It hinges on an API through which quiescence is requested, signaled, and then relinquished. Other work has attempted offline algorithms for choosing the best time point for reverting the storage state [17, 18].

Chronus [18] is a tool for automating the diagnosis of configuration errors caused by a state change. Applications run on VMs and periodic snapshots are taken of their storage state. When an error is encountered, Chronus automates the search for the time it was introduced. It uses a binary search among storage snapshots, while rebooting the VM and restarting the application on each one. Each such restart requires file system and application recovery, e.g., running *fsck*. While they suggest that a journalling file system will improve performance, it does not avoid the need for recovery altogether.

Assuming many recovery points are available, Sweeper [17] determines a recovery point that will provide a clean copy of application data. Ideally, a relatively clean copy reduces or eliminates the time spent on file system and application recovery. In contrast, by taking a *memory* checkpoint of the VM running the application, we avoid the need for file system and application recovery, operating system boot and application restart. Using an online algorithm, we consider what is the most meaningful time to take the checkpoint up front.

For computer systems, machine learning has been applied primarily in the domains of security and performance management, both outside the scope of this paper. Machine learning was also identified as a key component of autonomic computing [7]. But machine learning requires a steady stream of real-life data. To obtain it, one needs to either hook into existing implementations, which is not generalizable, or look at historical logs, which lacks timeliness. This is where virtualization can complement machine

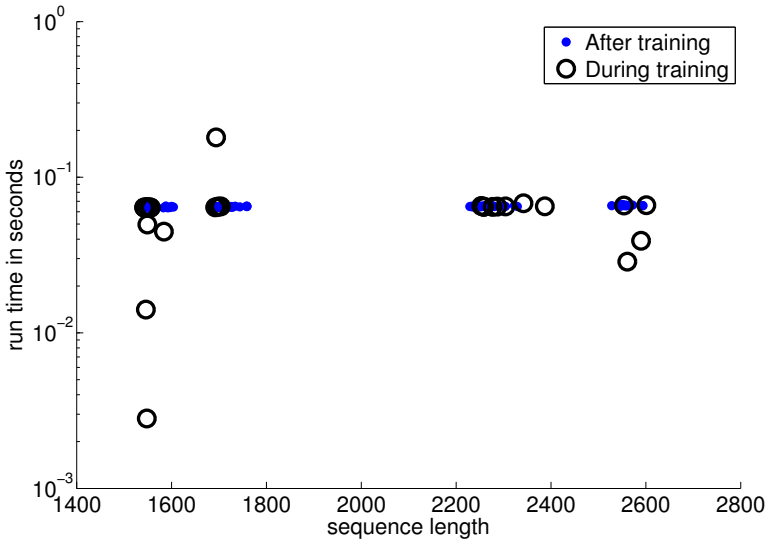

Figure 4: Run time, as a function of the sequence length (in blocks). Note the Y axis is on a logarithmic scale. This is data from the same 100 sequences used to generate Figure 3.

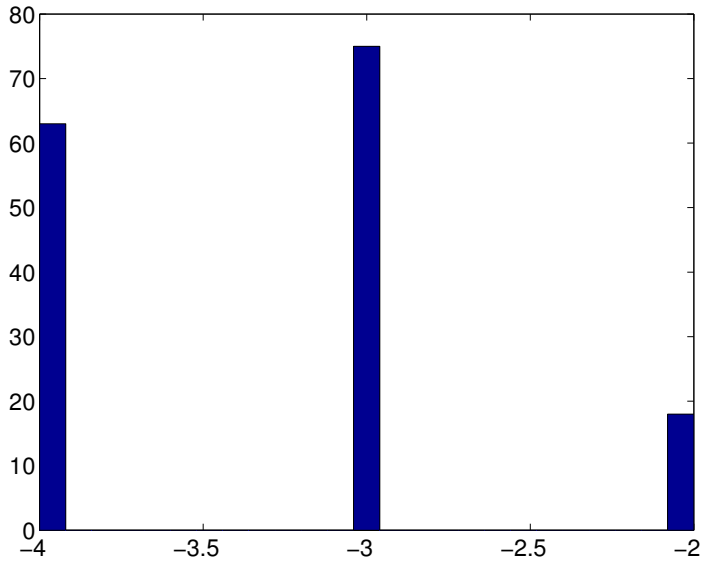

Figure 5: Histogram of detection errors, in seconds, for Linux. A normal boot sequence without DHCP delays takes about 22 seconds.

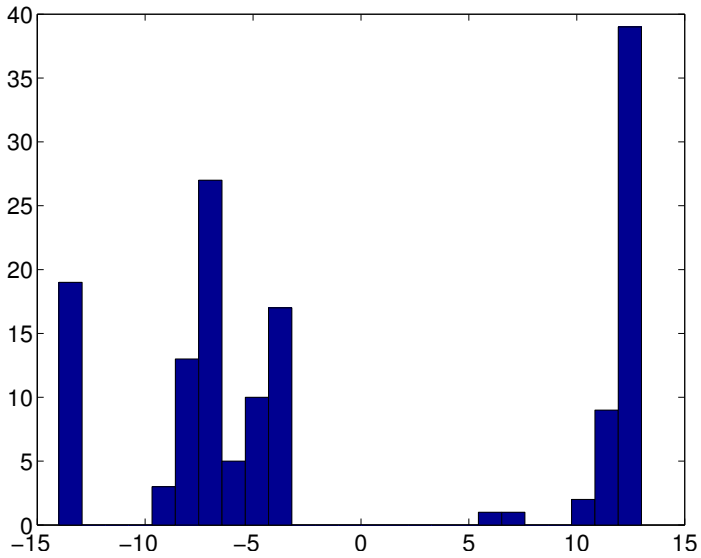

Figure 6: Histogram of detection errors, in seconds, for Windows. A normal boot sequence without DHCP delays takes about 35 seconds

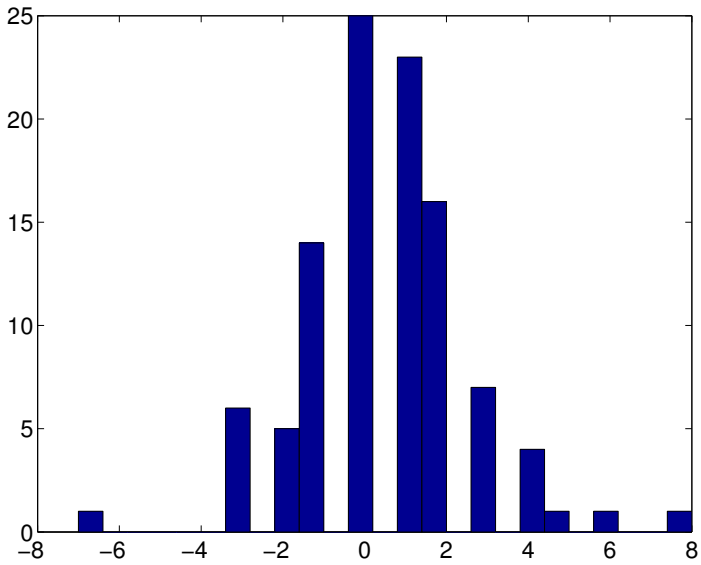

Figure 7: Histogram of detection errors, in seconds, for WAS. A normal boot sequence without DHCP delays takes about 60 seconds

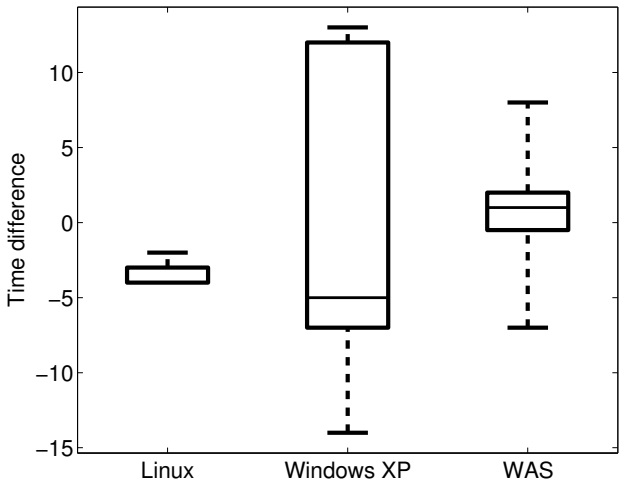

Figure 8: Summary of detection accuracy for various workloads. The box edges show the first and third quartile, and the middle line shows the median (for Linux, it aligns with the top border). The "whiskers" show the range of the data. Sample sizes are 156 for Linux, 146 for Windows XP, and 104 for WAS.

learning, by making available multiple data streams through various hypervisor APIs. Techniques from machine learning can then be used to analyze the data. This is the premise of the Vigilant [12] project, under which this work was performed.

5. 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 start-up sequence, either in addition or in place of the normal activity.

The assumption we make on the system are weak enough to enable deployment in a wide variety of installations, including virtual machines, diskless clients, CDP clients, and more. In each of these scenarios, the detection functionality is retrofitted without affecting or notifying the existing OS. This has important operational and economical implications.

Once a system is detected as failing to boot, possible remediation actions may include restarts, virus-scanning (if a significant change in file access is detected), or manual inspection. Conversely, a correctly-booted system can be checkpointed. Later, when the virtual machine needs to be restarted for whatever reason, the VM can be resumed from the checkpoint and immediately put to work, rather than booted cold.

Our method has been implemented and tested on both Windows and Linux virtual machines running a variety of workloads, and shown to be quite accurate. In most cases it detected the boot-termination event within 5 seconds of the event's occurrence.

We believe that monitoring the disk access stream out-of-band has potential much bigger than just boot-sequence detection. Some of our planned future work includes mining data from systems post-boot to obtain various kinds of operational advantages. As this capability is becoming more common, the potential benefits become too large to ignore.

6. 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] MATLAB. `http://www.mathworks.com/`.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, L. N. Bairavasundaram, T. E. Denehy, F. I. Popovici, V. Prabhakaran, and M. Sivathanu. Semantically-smart disk systems: past, present, and future. *SIGMETRICS Performance Evaluation Review*, 33(4):29–35, 2006.

[6] 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.

[7] J. O. Kephart. Research challenges of autonomic computing. In *ICSE '05: the 27th international conference on software engineering*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[8] 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.

[9] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *FAST '04: The 3rd USENIX Symposium on File and Storage Technologies*, pages 173–186, 2004.

[10] W. Norcutt. The iozone filesystem benchmark. `http://www.iozone.org/`.

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

[12] D. Pelleg, M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan. Vigilant: out-of-band detection of failures in virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(1):26–31, 2008.

[13] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *FAST'05: the 4th conference on USENIX Conference on File and Storage Technologies*, page 18, Berkeley, CA, USA, 2005.

[14] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *FAST '03: the 2nd USENIX Conference on File and Storage Technologies*, pages 73–88, Berkeley, CA, USA, 2003.

[15] B. Smaalders and P. Harman. libMicro. `http://www.opensolaris.org/os/project/libmicro/`.

[16] A. Tirumala and J. Ferguson. Iperf. `http://dast.nlanr.net/Projects/Iperf/`.

[17] A. Verma, K. Voruganti, R. Routray, and R. Jain. Sweeper: an efficient disaster recovery point identification mechanism. In *FAST'08: the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, Berkeley, CA, USA, 2008.

[18] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI'04: the 6th conference on Symposium on Opearting Systems Design & Implementation*, Berkeley, CA, USA, 2004.