Profiling the Operational Behavior of OS Device Drivers*

Constantin Sârbu†, Andréas Johansson‡, Neeraj Suri†, Nachiappan Nagappan§

Technische Universität Darmstadt†
Darmstadt, Germany
{cs,suri}@cs.tu-darmstadt.de

Volvo Technology Corporation‡
Göteborg, Sweden
andreas.olof.johansson@volvo.com

Microsoft Research§
Redmond, USA
nachin@microsoft.com

Abstract

As the complexity of modern Operating Systems (OS) increases, testing key OS components such as device drivers (DD) becomes increasingly complex given the multitude of possible DD interactions. If representative operational activity profiles of DDs within an OS could be obtained, these could significantly improve the understanding of the actual operational DD state space towards guiding the test efforts.

Focusing on characterizing DD operational activities, this paper proposes a quantitative technique for profiling the runtime behavior of DDs using a set of occurrence and temporal metrics obtained via I/O traffic characterization. Such profiles are used to improve test adequacy against real-world workloads by enabling similarity quantification across them. The profiles also reveal execution hotspots in terms of DD functionalities activated in the field, thus allowing for dedicated test campaigns. A case study on actual Windows drivers substantiates our proposed approach.

1 Introduction

In order to support a wide variety of hardware peripherals and new applications, commercial off-the-shelf (COTS) OSs are biased in their support for adaptability rather than service reliability. The OS interface to the hardware devices is represented by the *device drivers* (DD).

Recent research [1,2,5,6,18,21] has shown that unfortunately DDs constitute a dominant cause of OS failures. As DDs are often released without time-consuming exhaustive testing, they usually exhibit a higher defect density compared to the OS kernel [4]. Moreover, DDs are typically delivered as binaries constraining potential testing campaigns to black-box strategies.

On this basis an important consideration is the DD testing under "field" conditions. While multiple sophisticated static testing techniques for DDs exist [3,10,14], the choice of a relevant workload is key to exercise a DD in its actual operational domain. Although the operational profile of a DD is difficult to capture (and later to reproduce for testing), once obtained it can bring significant advantages over static testing techniques by identifying the triggered functionalities, their sequence and occurrence patterns. Subsequent test campaigns can primarily target code more likely to be executed in the field, therefore decreasing the time required to find the defects with high operational impact. Consequently, developers and system integrators are required to envision workloads that realistically mimic the manner in which the DD (or the whole system) will be used [22], i.e., the DD's operational profile. The more accurate the profile, the more effective a test campaign can be developed to test the DD state space.

Focusing on generating operational profiles to guide OS/DD testing, our approach is based on monitoring the interface between the OS kernel and the DDs. At this interface the flow of I/O requests is captured and used to build a state model of the DD. The state of a DD is represented by the set of DD functionalities in execution at a specified time instant. The transitions between states are triggered by I/O requests. The resulting behavioral model is used to discover execution hotspots in terms of most frequently used states of the system and to compare workloads.

Paper Contributions and Organization. This paper proposes and subsequently develops:

a) a profiling technique for DDs via I/O traffic characterization;
b) a set of occurrence- and time-based quantifiers for accurate DD state profiling;
c) a ranked set of states and transitions to assist execution hotspot discovery;
d) a state-based methodology for accurate workload activity characterization and comparison;

Additionally, being non-intrusive and based on black-box principles, our framework is portable and easy to implement in DD profiling scenarios where no access to the source code of either OS kernel, workload applications or target DDs is available.

*This research has been supported, in part, by Microsoft Research, NoE ReSIST and DFG TUD GK-MM.

The paper is organized as follows: Section 2 presents the related work, Section 3 discusses the system and DD models and describes a representation of a driver's operational profile. The quantifiers for characterizing DD runtime behavior are developed in Section 4. The experimental evaluation of our approach is presented in Section 5 along with a discussion on the findings in Section 6.

2 Related Work

SWIFI (Software-Implemented Fault Injection) is a technique widely used to assert the robustness of black-box DDs [5, 7, 10]. In industry, the main OS developers periodically release improved specifications and tools to minimize the risk of launching faulty DDs [3] and make efforts towards determining the "requisite" amount of testing [14, 15]. Unfortunately, in spite of considerable advancements in DD testing [1, 2, 4–6, 21], DDs are still a prominent cause of system failures.

Musa's work [12] on reliability engineering suggests that overall testing economy can be improved by prioritizing testing activities on the functionalities with higher impact on the component's runtime operation. Along the same lines, Weyuker [23, 24] recommends focusing on testing those functionalities with high occurrence probabilities. Results from the area of software defect localization show that faults have a tendency to cluster in certain parts of the OS code [4, 11]. Thus, a strategy aimed at clustering the code into functionality-related parts and then testing based on their operational occurrence is desired, especially when the resources allocated for testing are limited. Weyuker [22] underlines the necessity to test COTS components in their new operational environments even though they were tested by their producers or third-party testers.

Rothermel et al. [17] empirically examined several verification techniques showing that prioritization can substantially improve fault detection. Specifically, they believe that "*in a testing situation in which the testing time is uncertain (...) such prioritization can increase the likelihood that, whenever the testing process is terminated, testing resources will have been spent more cost effectively in relation to potential fault detection than they might otherwise have been*". Accordingly, this paper helps focusing testing onto the states of the driver-under-test based on their occurrence and temporal likelihoods to be reached in the field.

McMaster and Memon [9] introduced a statistical method for test effort reduction based on call stacks recorded at runtime. While their approach captures the dynamic program behavior, it is specific to single-threaded, user-space programs not being directly applicable to DDs (as they run in an arbitrary thread context). Leon and Podgurski [8] evaluated diverse techniques for test-case filtering using cluster analysis on large populations of program profiles, highlighting them as a useful basis for test effort reduction.

3 Driver State Definition via I/O Traffic

This section introduces our system model and the necessary technical background used to explain our state model for DDs. Figure 1 represents a typical computer system equipped with a COTS OS supporting a set of applications (the *workload*) that use services provided by the OS.

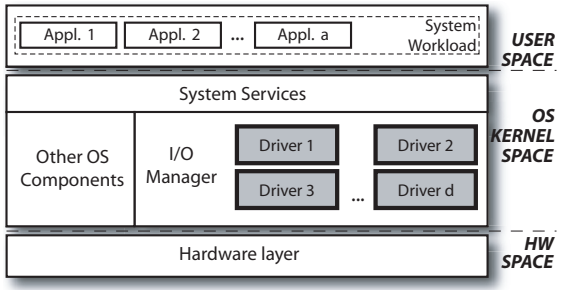

Figure 1. The considered OS system model.

This paper focuses on the communication interface between *I/O Manager* and *drivers* located within the OS kernel space. The *I/O Manager* is a collection of OS structures responsible for mediating the flow of I/O requests between the applications and the responsible DDs. A *driver* is an independent OS component handling the communication with one or more equivalent peripherals.

While our approach is applicable for generic DDs and OSs, we utilize Windows XP SP2 DDs as representative case studies for the proposed concepts. In Windows, DDs act on I/O requests initiated by the I/O Manager directly or on behalf of user applications. In this paper, the communication interface between the I/O Manager and the DDs is used to characterize the activity of a DD. At this level Windows uses the shared memory communication scheme specified by the Windows Driver Model (WDM) [15].

WDM-compliant DDs follow mandated rules governing design, initialization, power and memory management etc. Each DD must implement a minimal standard interface. Of major interest for our approach is that WDM requires the DD code to be internally organized in *dispatch routines*, each being responsible for handling a specific *I/O Request Packet* (IRP) type. The current WDM specification defines 28 IRP types, each of them associated with a specific operation (e.g., CREATE, READ, WRITE, CLOSE, etc.).

3.1 Driver Modes and Transitions

The state of a DD is characterized by the handled IRP requests. That is, a DD is idle from the initialization instant until the first IRP request is received. The DD is in the "*processing* IRP_a" state from the instant when IRP_a is received and until the DD announces its completion. We call this relaxed state "*driver mode*". In each mode, the DD executes a subroutine specific to the type of the received IRP,

as specified by the WDM [15].

Some IRPs are processed concurrently by the DD, i.e., processing IRP_b can start before IRP_a is completed (assuming that IRP_a was initiated before IRP_b). To the current extent of our experimental work we have never encountered situations when more than one IRPs of the same type are processed at once. Hence, in [19] we defined the *mode* of a driver D as follows:

Def. 1 (Driver Mode). *The mode of a driver D is defined by a tuple of predicates, each assigned to one of the n distinct IRP types supported by the driver:*

$$M^D : \ < P_{IRP_1}\ P_{IRP_2}\ ..\ P_{IRP_i}\ ..\ P_{IRP_n} >,\ where$$

$$P_{IRP_i} = \begin{cases} \textbf{\textit{1}},\ \textit{if D is currently}\ \textbf{performing}\ \textit{the functionality triggered by the receival of}\ IRP_i \\ \textbf{\textit{0}},\ \textit{otherwise} \end{cases}$$

As the driver mode is a binary tuple of size n, the total state space size (the total number of modes) for the driver D in our model is 2^n.

Def. 2 (Operational Profile). *The* operational profile *(OP) of a DD with respect to a workload is the set of modes visited in the time interval spanning the workload execution.*

Our previous experimental investigations showed that the size of the OP is a small fraction of the total set of modes ($N_{OP} \ll 2^n$), irrespective of the workload.

As the I/O Manager serializes both the sending and receival of IRPs, only one bit changes at a time. Thus, a DD can only switch to modes whose binary tuples are within Hamming distance of 1 from the current mode. Consequently, there are n transitions possible from each mode, implying the total number of transitions in our model to be $n \cdot 2^n$. As the number of transitions traversed for a workload represents only a subset of the total transitions ($T_{OP} \ll n \cdot 2^n$) [19], the actual state space that needs consideration is significantly reduced. If needed, such a DD's OP can also be used to proactively discover new modes by forcing the leaf-modes to traverse non-transited edges (e.g., for robustness testing).

3.2 Characterizing a Driver's Behavior

In our prior work we experimentally identified the OP of the serial port DD provided with Windows XP SP2 [19]. The results revealed that the reached modes and transitions and the obtained OP are *consistent* across different runs. Moreover, the OP of the studied DD has a very *small footprint* (only 1.6% of the modes were visited and 0.3% of the transitions were traversed). Though small and stable, the applicability of the OP for operational profiling purposes is limited as it divides the modes and transitions into only two subsets (i.e., gives only binary information: visited and non-visited). Unfortunately, this is insufficient for a proper characterization of the runtime activity as the ability to distinguish among the visited modes and transitions is missing.

In the next section we enhance the captured operational profiles by introducing additional metrics for an accurate characterization of the driver's runtime behavior. We start from the hypothesis that the higher detail level of the OP quantification permits discovery and assessment of the existing execution hotspots in driver's code.

4 Quantifiers for Runtime Behavior

Accurate methods for characterizing the operational behavior of a software (SW) component are desirable for establishing effective testing methods. Hence, this section presents a set of quantifiers for the operational phase of a DD that are developed for differentiating among the modes and the transitions of the OP. Using them, one can observe and analyze the relative frequencies of the visited modes and transitions, revealing execution hotspots.

The metrics presented in this section are useful as they provide accurate workload characterization from the DD's perspective. For instance, capturing the driver's activity in the field can be used for workload assessments, usage/failure data collection or post-mortem debugging. Moreover, different workloads can be compared to statistically reveal the DD modes with higher probability of being reached in the field.

The developed metrics have a statistical meaning in the context of the workload for which they were assessed. Within this perspective, the OP can be used to detect a failure of a DD by observing a population of runs of the selected workload and finding the OP which deviates from the rest of the runs in terms of one (or multiple) quantifiers. Section 5.3 illustrates the workload comparison procedure enabled by our approach.

For testing purposes, our measures can be used to compare from a quantitative viewpoint the effects of single test cases or test suites on the driver-under-test. If the intent of testing is higher DD code coverage at lower costs, one can select the test cases (suites) having the highest coverage in terms of reached driver modes. Hence, while still reaching the same DD functionalities, the size of test case pool can be reduced to a least necessary minimum. Section 5.4 provides an illustration of the test space reduction of our method.

4.1 Occurrence-based Quantifiers

Two important characteristics of the runtime behavior of a DD are the *occurrence weights* for both modes and transitions. They reflect the DD's preference to visit the mode (or transition) to which these quantifiers are bound. To express them, we first define as prerequisite notions the transition and mode *occurrence counts* for a given workload w. We define a driver's OP as a digraph with the set of modes $M = \{M_1^D, M_2^D, \ldots\}$ as vertices and the set of transitions $T = \{t_1, t_2, \ldots\}$ as edges. Each transition from T maps to

ordered pairs of vertices (M_i^D, M_j^D), with $M_i^D, M_j^D \in M$, $i \neq j$ and the modes M_i^D and M_j^D are within a Hamming distance of 1 from each other.

Def. 3 (**Transition Occurrence Count**: $TOC_{t_{i,j}}$)**.** *The occurrence count for transition $t_{i,j} \in T$, originating in mode M_i^D and terminating in mode M_j^D ($M_i^D, M_j^D \in M$ and $i \neq j$) is the total number of recorded traversals from mode M_i^D to mode M_j^D.*

Def. 4 (**Mode Occurrence Count**: MOC_j)**.** *The occurrence count of mode $M_j^D \in M$ is the number of times mode M_j^D was visited during the execution of workload w.*

$$MOC_j = \sum_{i=1}^{N_{OP}} TOC_{t_{i,j}} \tag{1}$$

Note that both the occurrence counters expressed above are defined for the duration of the workload w. Counter variables associated with each mode and transition can be used to store their values. The TOC and MOC counters are utilized to develop subsequent quantifiers accurately specifying the operational behavior of DDs, namely the *mode occurrence weight* and the *transition occurrence weight*.

Def. 5 (**Mode Occurrence Weight**: MOW_i)**.** *The occurrence weight of mode $M_i^D \in M$ represents a quantification of a driver's preference to visit the mode M_i^D relatively to all other sojourned modes of the OP (N_{OP}), for the workload w.*

$$MOW_i = \frac{MOC_i}{\sum_{i=1}^{N_{OP}} MOC_i} \tag{2}$$

This metric is similar to the metric used for development of operational profiles for building reliable SW components proposed by Musa [12]. In contrast to [12], our quantifier is specific to profiling the runtime behavior of kernel-mode DDs and its significance is coupled with the specific workload for which it was computed. If the chosen workload accurately mimics the manner in which the DD is used in the field, the obtained mode quantifiers accurately express the field conditions.

Using this metric in profiling the runtime behavior of a DD helps building test priority lists. For instance, the modes with higher MOW value represent primary candidates for early testing, as higher values of this quantifier indicate the functionalities of the DD which are most frequently executed. For the *idle mode* (i.e., when the DD is not executing any IRP-related activity) this measure indicates the percentage of mode sojourns that put the DD in an idle state, i.e., waiting for IRP requests.

Similarly to MOW but referring instead to the transitions between modes, we define the *transition occurrence weight* for each traversed transition belonging to the DD's OP for a given workload.

Def. 6 (**Transition Occurrence Weight**: $TOW_{t_{i,j}}$)**.** *The occurrence weight of transition $t_{i,j} \in T$, originating in mode M_i^D and terminating in mode M_j^D ($M_i^D, M_j^D \in M$ and $i \neq j$) is the quantification of driver's preference to traverse the transition $t_{i,j}$ when leaving the mode M_i^D.*

$$TOW_{t_{i,j}} = \frac{TOC_{t_{i,j}}}{MOC_i} \tag{3}$$

Thus, the occurrence weight associated with transition $t_{i,j}$ indicates the probability that this transition is actually followed when leaving the mode M_i^D. Note that the probability of following a certain transition depends on the current mode. This information is relevant for estimating which mode is to be visited next, given that there is a one-hop transition in the OP between the current mode and the one whose reachability is to be calculated.

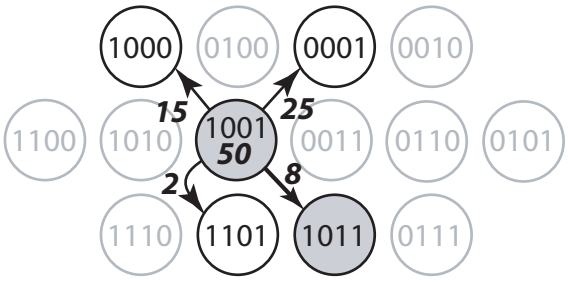

Figure 2. An example for calculating $TOW_{t_{1001,1011}}$.

For instance, let us consider the situation depicted in Figure 2, where only the modes and the outgoing transitions of interest are shown, together with their TOC values as edge labels. The mode `1001` is current and the $MOC_{1001} = 50$. $TOC_{t_{1001,1011}} = 8$. Therefore, $TOW_{t_{1001,1011}} = \frac{8}{50} = 0.16$. This indicates that the transition between `1001` and `1011` has been traversed 16% of the times the mode `1001` was left.

4.2 A Time-based Quantifier for Modes

To increase the accuracy of DD profiling not only the spatial dimension should be considered but also the temporal dimension. We consider time not just an artifact of the device's inherent slowness but a very important aspect of the computation, as longer execution time reveals more defects (shown by many defect estimators in the field, e.g., the Musa-Okumoto model [13]). Therefore, we introduce a quantifier that accounts for the relative amount of time spent by the DD executing in each mode. As we consider the transitions between modes as instantaneous events, defining a corresponding temporal measure for edges is superfluous.

An indication of the overall time spent by the DD in each mode can reveal valuable information about the latencies of various IRP-related activities of a DD, being helpful for guiding a testing campaign. If the DD spends a relatively large amount of time in a certain mode, that mode can be considered important for a subsequent testing campaign although the respective mode has a very low occurrence count (and, implicitly MOW). For instance, the DDs managing disk or tape drives spend large amounts of time in modes associated with `READ` or `WRITE` operations, irrespective of their sojourn rate. To capture this behavior we introduce a new OP quantifier, the *mode temporal weight*, to be used in conjunction with the mode occurrence weight for a multivariate characterization of driver modes.

Def. 7 (**Mode Temporal Weight**: MTW_i)**.** *The temporal weight of the mode $M_i^D \in M$ is the ratio between the amount of time spent by the driver in mode M_i^D and the total duration of the workload w.*

4.3 A Compound Quantifier for Modes

An accurate characterization of the runtime behavior of a DD needs to consider both the occurrence and time-based metrics, on a stand-alone basis and in combination. In order to facilitate combinations of the two metrics, we propose a compound measure that captures these dimensions of the profiled DD, namely the *mode compound weight*.

Def. 8 (**Mode Compound Weight**: MCW_i)**.** *Let λ be a real number, $0 \leq \lambda \leq 1$. The compound weight of a mode $M_i^D \in M$ is given by the expression:*

$$MCW_i(\lambda) = \lambda MOW_i + (1 - \lambda)MTW_i \qquad (4)$$

By varying λ, this measure can be biased towards either occurrence or temporal dimension, to best accommodate the needs of the tester. For instance, to emphasize the temporal aspect λ should take values closer to 0, while the occurrence dimension is highlighted by values of λ that approach 1.

5 Experimental Evaluation

To validate the profiling approach and the associated metrics, we conducted a series of experiments that investigate the following research questions:

- **Q1:** Can the metrics be used for comparing the effects of different workloads on DDs?
- **Q2:** How can the OP quantifiers be applied in practice for test space reduction?

To answer these questions, the following subsections show how the rankings based on execution quantifiers are obtained (Section 5.2), how workloads are compared among each other (Sections 5.3) and how are our OPs usable for test space reduction (Section 5.4).

To capture the IRP flow, we have built a lightweight "*filter driver*" interposed between the I/O Manager and a DD of our choice (Figure 3). This mechanism is widely used by many OSs for modifying the functionality of existing DDs or for debugging purposes. Our filter driver acts as a wrapper for the monitored DD, only logging the *incoming* (from I/O Manager to DD) and *outgoing* (from DD to I/O Manager) IRPs. The IRP communication flow is then forwarded unmodified to the original recipient. As for each IRP only a call to a kernel function is needed for logging it, we expect the computation overhead of our filter driver to be marginal.

Interposing our filter driver between the I/O Manager and a selected DD is done using the standard driver installation mechanisms offered by Windows. Hence, it is completely non-intrusive and does not require knowledge about the wrapped DD. The insertion and removal of the filter driver require only disabling and re-enabling the target DD but no machine reboot. Moreover, due to its conformance to WDM, we used (sans modifications) the same filter driver to monitor all DDs whose runtime behavior were investigated in this paper.

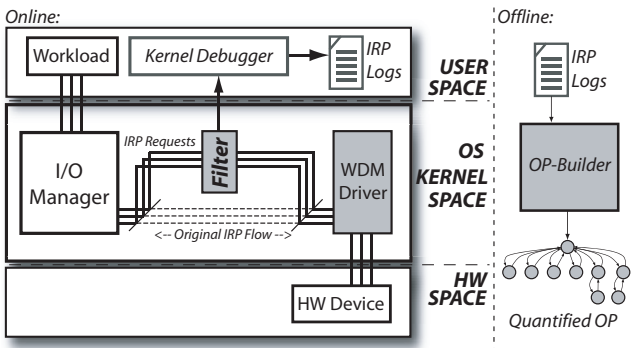

Figure 3. The experimental setup; the online (monitoring) and offline (analyzing) phases.

To compute the OPs for different DDs, we have built a tool (*OP-Builder*) that processes the logs and outputs the DD's OP, together with all runtime quantifiers. The figures 4–6 are obtained using directly the outputs of the *OP-Builder* tool. For our experiments we utilized a Pentium4@2.66Ghz machine with 512Mb of DDRAM, equipped with Windows XP Professional 5.01.2600 SP2. To build the filter driver we have used Windows Server 2003 DDK. For logging the kernel messages sent by the filter driver we have used Sysinternal's DebugView tool.

5.1 Studied Drivers and Workloads

In this paper we present a systematic evaluation of five diverse types of WDM-compliant DDs: a serial port driver (`serial.sys`), a CDROM driver (`cdrom.sys`), an ethernet card driver (`sysnic.sys`), a floppy driver (`flpydisk.sys`) and a parallel port driver (`parport.sys`). All DDs are provided (and digitally signed) by Microsoft, except the `sysnic.sys`, which is

Table 1. The workloads utilized to exercise the DDs and the overall experimental outcomes.

Driver	Short	Benchmark	Generated IRPs		Visited		Benchmark Description (Duration)
			Issued	Types	Modes	Edges	
cdrom.sys	**C1**	BurnInTest-Audio	1336	3 out of 9	4	6	Audio CD test mode (7'40")
	C2	BurnInTest-Data	71012	2 out of 9	3	4	Data CD read and verify mode (1'3")
sysnic.sys	**E1**	BurnInTest	2480	4 out of 28	5	8	TCP/UDP full duplex (5'3")
	E2	DevMgr-Disable	6	1 out of 28	2	2	Disable ethernet from Device Manager(1")
	E3	DevMgr-Enable	114	6 out of 28	7	12	Enable ethernet from Device Manager (2")
flpydisk.sys	**F1**	BurnInTest	2946	5 out of 6	6	10	Various pattern read and write (5'3")
	F2	Sandra Benchmark	19598	5 out of 6	9	16	Performance benchmark filesize: 512b - 256kb (23'40")
	F3	DC2	50396	4 out of 6	5	8	MS Device Path Exerciser (32")
	F4	DevMgr-Disable	10	1 out of 6	2	2	Disable drive from Device Manager (0.01")
	F5	DevMgr-Enable	400	3 out of 6	4	6	Enable drive from Device Manager (17")
	F6	F1 – F5, sequentially	63396	5 out of 6	6	10	Sequential execution of F1–F5 (31')
	F7	F1 || F2 (run I)	12298	5 out of 6	15	34	Concurrent execution of F1+F2 (18'40")
	F8	F1 || F2 (run II)	21884	5 out of 6	15	34	Concurrent execution of F1+F2 (27'50")
parport.sys	**P1**	DC2	48530	6 out of 6	7	12	MS Device Path Exerciser (5.7")
serial.sys	**S1**	BurnInTest	11568	6 out of 6	9	16	COM1 loop-back test (5')

provided by the SiS Corporation. To properly exercise the chosen DDs, we selected a set of benchmark applications generating comprehensive, deterministic workloads for the targeted DDs. For each experiment we analyzed the collected logs, we constructed the OP graphs as in Figure 4 and we calculated the OP quantifiers defined in Section 4.

Besides commercial benchmarks that test the performance and reliability of the peripherals under various conditions, it is worth mentioning that we have also additionally used the *Device Path Exerciser* (DC2) tool. DC2 is a robustness testing tool that evaluates if a DD submitted for certification with Windows is reliable enough for mass distribution. It sends the targeted DD a variety of valid and invalid (not supported, malformed etc.) I/O requests to reveal implementation vulnerabilities. DC2 requests are sent in synchronous and asynchronous modes and in large amounts over short time intervals to disclose timing errors. In our experiments we exercised the parallel port and the floppy disk driver with a comprehensive set of the DC2 tests.

The results of experimenting with the chosen DDs are summarized in Table 1. Due to space restrictions we limit the scope to a detailed study only for the `flpydisk.sys` driver. In the full spectrum of our experiments all the other mentioned DDs also showed the effectiveness of our OP profiling method [20].

5.2 Case Study: the flpydisk.sys Driver

Figure 4 depicts the OP of the `flpydisk.sys` driver, as exercised by the `F2` workload (see Table 1). Each node contains the mode name, MOW and MTW values (the latter in square brackets). Similarly, the TOW value of each transition is attached to the respective directed edge. For simplicity, the occurrence counters of modes (MOC) and transitions (TOC) are not shown in the graph.

In Figure 4 the darker gray hemispheres represent higher MOW or MTW values, revealing the DD functionalities frequently executed and the ones with longer execution times, respectively. The mode `000000` was predominantly visited

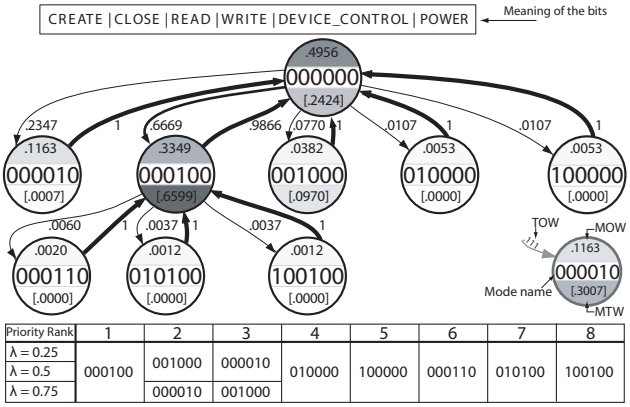

Figure 4. The OP for F2 and three prioritization cases. Darker shades and thicker lines indicate higher weights (execution hotspots).

under the workload, indicating that the DD was idle almost 25% of the time (low driver load). Most of the time (66%) was spent by the DD executing the functionality of the mode `000100`. This result is intuitive, as this mode is associated with the slow WRITE operation of the floppy disk drive.

Depending on the testing purpose, the balance factor λ (see Eq. 4, MCW) can be tuned to guide test prioritization. Varying λ from 0 to 1 is equivalent to move emphasis from MTW to MOW. The table in Figure 4 represents the test priority ranking based on MCW of the modes when λ is 0.25, 0.5 and 0.75 (rank 1 has the highest priority). For instance, when $\lambda = 0.5$ (equal balance between MOW and MTW weights is desired), the mode `000100` is identified as a primary candidate for testing, followed by `001000` and `000010`. In contrast, a $\lambda = 0.75$ (when MOW dominates MTW) keeps the same mode on the first priority rank but swaps the order of the second and the third modes. All other

modes keep their ranking irrespective of the value of λ.

Our approach also reveals the mode sojourn sequences, enabling the observation of the temporal evolution of the DD. This representation exposes the functionalities in execution at any instant enabling testers to reproduce the IRP sequence that brings the DD in a mode of interest (i.e., to create effective test cases).

5.3 Comparing Driver Runtime Behaviors

Continuous post-release service provision is problematic mainly due to the limited capacity of the in-house testing to accurately reproduce the entire spectrum of possible workloads. To estimate the difference between the behavior seen during in-house testing and field behavior, a comparison of the operational profiles can be performed. Ideally, a minimal difference ensures post-release test adequacy [22], though measuring it is non-trivial. It requires detailed knowledge about the field workload and a set of metrics to express the deviation. Assuming that several OPs of the DD are available, our runtime behavior quantifiers can be used to outline their relative deviation.

We present a comparative study of two workloads for the floppy disk driver, workloads F7 and F8 (see Table 1). For F7, we first started the F2 workload, followed after few minutes by the start of F1. For F8, we swapped the start order. Both F1 and F2 are performance benchmarks that create, read, write and delete files on the floppy disk. Both F1 and F2 complete successfully when run concurrently despite the fact that F7 is 9 minutes and 9586 IRPs shorter than F8.

We observed that F7 and F8 share the same OP structure (exactly the same modes and transitions were visited – see [20]). We note that considering only the structure of the OP graphs when comparing workloads is not effective for runtime behavior profiling, as the level of provided detail is limited. For instance, it is impossible to differentiate between a workload executing multiple READ operations and a workload accessing this operation only once. This fact recommends using the OP quantifiers for a more accurate comparison.

Figure 5 shows a side-by-side representation of the MCW ($\lambda = 0.5$) values for each mode, as generated by the F7 and F8 workloads. Similarly, Figure 6 depicts the TOW values side-by-side for each traversed transition. The values above the bars represent the absolute difference between the MCW values of each mode, and between TOW values for each transition. In both figures the small difference between the two OPs is apparent, despite of an almost double amount of issued IRPs for F8 relative to F7. The largest deviations among MCW values are for modes `000010` (0.024) and `001100` (0.015), an indication that F7 executed more READ + WRITE operations and less DEVICE_CONTROL in contrast with F8.

Even though the same modes and transitions were visited under both workloads, the distance between TOW values show a wider distribution than the MCW values. Figure 6 indicates the transition `001000>000000` (the bars pointed by the arrow in Figure 6) having the largest difference between TOW values of F7 and F8, with 0.280 bias towards F8. This transition is traversed when the READ operation finishes. A closer inspection reveals that F7 compensates this difference with higher TOW values of the other transitions originating in `001000`, i.e., transitions to the modes `001100`, `001010` and `101000`. As these modes are located on a lower level than `001000`, this reveals a certain tendency of F7 to start the concurrent execution of WRITE, DEVICE_CONTROL or CREATE while still running READ operations. This behavior might disclose potential problems related to concurrency handling or indicate places for optimizations.

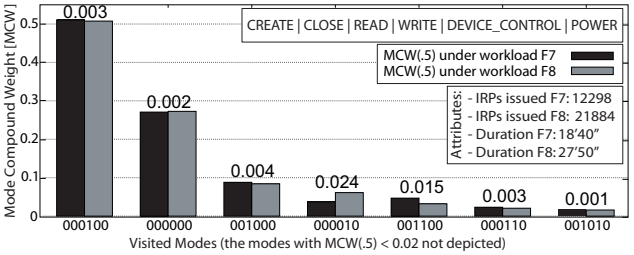

Figure 5. MCW comparison for F7 and F8.

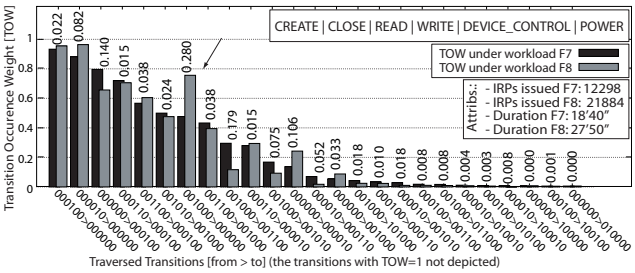

Figure 6. TOW comparison for F7 and F8.

Therefore, for comparing the effects of several workloads on a DD, the Euclidean distance between MCW values of each mode or between TOW values of each transition can be evaluated. Moreover, if the distance is smaller than a given threshold, then the compared workloads are considered equivalent. This constitutes valuable feedback for ascertaining the adequacy of a testing campaign versus operational usage.

We use *multidimensional scaling* (MDS) plots to graphically display the distances between the workloads used to exercise the `flpydisk.sys` driver. MDS is a statistical technique used to visually express the degree of similarity (or dissimilarity) among objects belonging to a population, each object being characterized by a set of attributes. For each attribute a distance matrix is computed among the ob-

jects. To evaluate the similarity among the objects, the distance matrices associated with each object attribute are aggregated in a weighted average. The MDS plot is computed using this average. For the MDS plot depicted in Figure 7 we have used the Euclidean distance among the MCW values of the corresponding modes visited by each workload. Similarly, Figure 8 is computed using the Euclidean distance among the TOW values of the corresponding transitions for each workload. For instance, the closeness between two points in Figure 7 indicates that the associated workloads have visited the same modes, generating similar MCW values for each of them.

Figure 7 shows the MDS plot of the workloads F1 - F8, where the attributes of the objects are the MCW values of every sojourned modes of the OPs (i.e., 15 modes, see Table 1), with a $\lambda = 0.5$. If a workload did not visit a certain mode in our experiments, the MCW value of that attribute is zero. The MDS plot in Figure 7 reveals that the DC2 is most similar to F4 and F5, two workloads that are representative for the operations performed when the floppy driver is loaded and unloaded from the OS. The comparison between the modes of the F7 and F8 presented in Figure 5 is confirmed by the MDS plot, as the points associated with the two workloads are very close to each other.

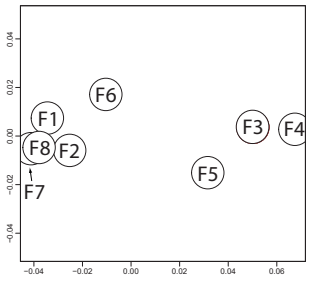

Figure 7. The distance among workloads (modes only).

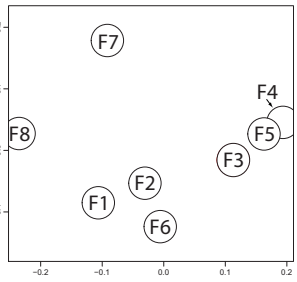

Figure 8. The distance among workloads (edges only).

To examine how the workloads compare from the perspective of the traversed edges, in Figure 8 we have considered as object attributes the TOW values of every transition of the OPs. Here, the F7 and F8 are farther away from the rest of the workloads. This effect is explained by the fact that F7 and F8 actually traverse all the 34 transitions that act as attributes of the objects in this plot, while the rest of the workloads only traverse a small subset of them.

As we mentioned before, for the MDS plots in Figures 7 and 8 we assigned equal weights to modes and transitions, respectively. Actually, to accurately ascertain how close a testing campaign is from the manner the DD is exercised under realistic workloads, one should assign heavier weights to the modes and to the transitions carrying the most of interest when the inter-object distances are computed.

Therefore, using the DD state quantifiers introduced in this paper, multidimensional scaling analysis can be successfully used on the data provided by our OPs to quantify the relative similarities among several workloads (for instance, field workloads vs. in-house testing workloads). This reveals new possibilities for statistical measurement of test coverage in terms of execution patterns.

5.4 Test Space Reduction Aspects

To evaluate the test-space reduction enabled by our current profiling approach, we compared it with our previous method [19], considering both modes and transitions that need be covered by a testing campaign. Table 2 lists the overall improvement introduced by the current approach.

First-pass reduction: For the Figures 9 and 10 we have selected for each DD the workload that exercised in our experiments the largest set of modes and transitions, respectively. For instance, for the floppy disk driver we selected the workload F7, as it issues IRPs belonging to 5 distinct types. This indicates a *theoretical state space* size of $2^5 = 32$ modes and $2 \cdot 2^5 = 64$ transitions. Out of this set, only 15 modes (46.87%) and 34 transitions (53.12%) were visited (see Table 1, columns 5–7). In Figures 9 and 10, we call this early reduction step *first-pass reduction*.

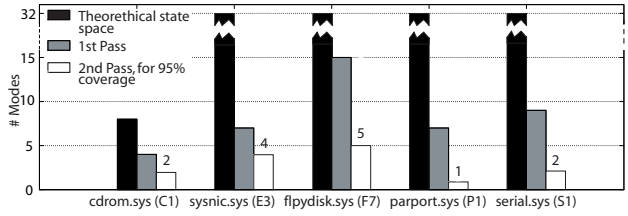

Figure 9. Test space reduction for modes.

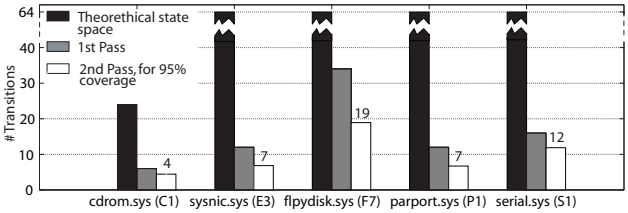

Figure 10. Test space reduction (transitions).

While the first-pass reduction solely divides the theoretical state space into two classes (visited and non-visited), this paper goes beyond that by utilizing the newly introduced execution quantifiers for modes and for transitions. They permit a finer differentiation among the visited modes (and among the traversed transitions), offering subsequent testing campaigns a richer insight about the modes and transitions that need consideration. In Figures 9 and 10 this step is called *second-pass reduction* and it is based on the relative ranking among modes and transitions, respectively. The process of obtaining such rankings is discussed in Section

5.2 and depicted by Figure 4.

Second-pass reduction using priority rankings: To explain the mechanism of the second-pass reduction, we introduce a threshold T specifying the desired test coverage level. Onwards, the word "coverage" means the coverage of the modes and transitions in our model. In relation with the ranking of modes (or transitions, respectively), T gives the reduction of the second-pass. For example, if a test coverage of 95% is desired for the visited modes under the workload F7, then they are selected from a list ranked using their MCW values. Figure 11 depicts the cumulative MCW value of the modes visited under F7. The modes are ordered in decreasing MCW order, from left to right. Therefore, when the example coverage goal is 95% of the visited modes, the first five leftmost modes (marked in the figure) are selected. This gives a reduction of 66.66% compared to the first-pass presented in [19]. When for instance a stricter 99% coverage of visited modes is desired, three more modes are selected, still giving a 46.66% reduction relative to the first-pass.

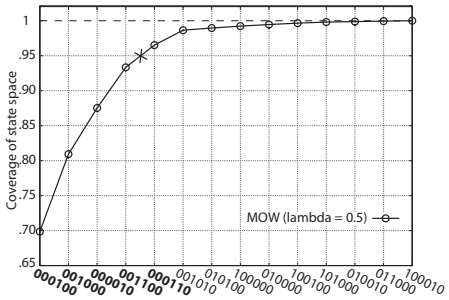

Figure 11. Cumulative coverage for F7; modes are ordered in decreasing MOW order from left to right.

Figure 11 indicates that the most visits are concentrated to a very small number of modes. This trend also holds for the transitions, indicating a high reduction in modes and transitions selected by the second-pass, also when the desired coverage is high. Therefore, this mechanism of selecting the modes of interest based on the rankings given by the execution quantifiers can be tuned using the threshold T to best fit coverage goals. Figures 9 and 10 depict the second-pass reduction of modes (and transitions, respectively) for 95% testing coverage. Table 2 lists the reduction for each of the workloads considered in Figures 9 and 10.

Table 2. Test space reductions of the second-pass relative to the first-pass ($T = 95\%$).

Quantifier	C1 [%]	E3 [%]	F7 [%]	P1 [%]	S1 [%]
MCW	50.00	42.85	66.66	85.71	77.77
TOW	33.33	41.66	44.11	41.66	25.00

Hence, when testing resources are limited, we believe that the testing effort can efficiently be scoped for the desired coverage level in order to first cover the modes and transitions associated with high likelihoods to be reached in the field. Assuming that the test effort is equally distributed among the visited modes, this result indicates that a significant reduction in the amount of testing is possible, without affecting its adequacy. While the test case creation and prioritization are out of the scope of this paper, we believe that existing test methods can effortlessly take advantage of the new insights offered by our profiling methodology.

5.5 Experiment Issues and Lessons Learnt

Our experiments showed that some workloads issue large amounts of IRPs per time unit, an example being the DC2 workload. This puts high pressure on the monitoring mechanisms, introducing the risk that important behavioral information may be lost. Our tracking filter driver has been built with great care to ensure that no IRPs are lost, and we have verified offline all traces to ensure that no invalid transitions are made (two bits in the mode are changed) and that all IRP pairs are matched. The same issue holds for the kernel debugger (see Figure 3), which is unable to track all IRPs when the arrival rate is extremely high. A discussion with the developers of the debugger revealed that the IRPs are stored in a temporary buffer. When the buffer is full, new IRPs are simply dropped. We are currently working on an improved version of the logging mechanism, which circumvents the problem. Note that for data presented in Table 1 is complete with respect to the tracked IRPs.

6 Summary and Discussion

Based on a non-intrusive state capture framework, our efforts provide accurate metrics and guidance for profiling and quantifying the runtime behavior for kernel-mode DDs. The presented experiments show the applicability of our approach to capturing the runtime behavior of actual DDs belonging to diverse classes. Summarizing, the empirical investigation performed in this paper shows the utility and the relevance of our state quantifiers for DD testing, as they:

- **reveal driver state sojourn patterns** without access to source code of the DDs;
- **enable workload characterization and comparison** for selecting the most appropriate workload for in-house testing;
- **assist test prioritization decision** based on quantified DD runtime profiles;
- **reduce the space size for testing activities** based on "tunable" coverage scenarios.

In this respect, our metrics do provide a useful quantification of the runtime behavior of a DD. As our OP quantifiers are statistical in nature, their relevance is directly proportional to the completeness of the workload used for ob-

taining them. Therefore, the choice of the workload used when building the OP is important and the monitoring phase should be long and diverse enough such that relevant behavior is captured. Such behavior is best captured if the monitoring is performed in the field, for instance during beta-testing or post-release monitoring. We have chosen commercial benchmarks to exercise our DDs as we believe they generate a mix of I/O requests with enough variety to be representative for the way DDs are used in the field.

Besides DD profiling, our state-aware DD monitoring method is relevant for failure data collection as well. As many OS failures are caused by faulty DDs, adding state information to the collected crash reports can aid debugging by revealing the DD state history.

From the testing perspective, our approach primarily provides a method to gain insight into the behavior of the driver-under-test at runtime. It is not intended as a tool for finding bugs. The main purpose is to quantify a DD's state sojourn behavior in order to guide testing towards certain subroutines, but it doesn't reveal where the fault lies. By prioritizing the test activities using the metrics proposed in this paper, testers can increase the likelihood of finding sooner the existing faults.

As the DDs considered for case studies are specific to Windows-family OSs, we believe that the presented methods can already be used with Vista kernel-mode DDs as they follow a super-set of the WDM specifications (the KMDF - Kernel-Mode Driver Framework [16]). Given the empirical nature of the presented experiments, we are aware that the validity of the results is currently limited to WDM-compliant DDs. Therefore, we are currently studying the porting of our approach to other OSs.

Using the lessons learnt from our DD monitoring approach, we are also considering to develop a technique for tuning an existing DD test tool to primarily cover the execution hotspots. The selection of test cases will consider the information obtained from a prior driver profiling phase in order to reduce the overall testing overhead. This will also help identifying modes which are insufficiently tested and assess the impact of this fact on DD robustness, together with an investigation of the possibility to correlate known Windows failures to DD OPs.

References

[1] A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the linux kernel. In *Proc. DSN*, pp. 867–876, 2004.

[2] J. Arlat, J.-C. Fabre, and M. Rodriguez. Dependability of COTS microkernel-based systems. *IEEE Trans. on Computers*, volume 51, issue 2:138–163, 2002.

[3] T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *IFM*, pp. 1–20, 2004.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. *Proc. SOSP*, pp. 73–88, 2001.

[5] J. Duraes and H. Madeira. Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. *IEICE Trans. on Information and Systems*, 86(12):2563–2570, 2003.

[6] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows xp kernel crash analysis. In *Proc. LISA*, pp. 12–22, 2006.

[7] A. Johansson and N. Suri. Error propagation profiling of operating systems. In *Proc. DSN*, pp. 86–95, 2005.

[8] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proc. ISSRE*, pp. 442–453, 2003.

[9] S. McMaster and A. M. Memon. Call stack coverage for test suite reduction. In *ICSM*, pp. 539–548, 2005.

[10] M. Mendonca and N. Neves. Robustness testing of the Windows DDK. In *Proc. DSN*, pp. 554–564, 2007.

[11] K.-H. Möller and D. Paulish. An empirical investigation of software fault distribution. In *Proc. METRICS*, pp. 82–90, 1993.

[12] J. D. Musa. *Software Reliability Engineering.: More Reliable Software Faster and Cheaper*. 2nd edition, 2004.

[13] J. D. Musa and K. Okumoto. A logarithmic Poisson execution time model for software reliability measurement. In *Proc. ICSE*, pp. 230–238, 1984.

[14] N. Nagappan, L. Williams, J. Osborne, M. Vouk, and P. Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. In *Proc. ISSRE*, pp. 10–18, 2005.

[15] W. Oney. *Programming the MS Windows Driver Model*. Microsoft Press, 2003.

[16] P. Orwick and G. Smith. *Developing Drivers with the Windows Driver Foundation*. Microsoft Press, 2007.

[17] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Trans. on Software Engineering*, 26:929–948, 2001.

[18] D. Simpson. Windows XP embedded with service pack 1 reliability. *http://msdn2.microsoft.com/en-us/library/ms838661.aspx*, Microsoft Corp., 2003.

[19] C. Sârbu, A. Johansson, F. Fraikin, and N. Suri. Improving robustness testing of COTS OS extensions. In *ISAS*, Springer LNCS 4328, pp.120–139, 2006.

[20] C. Sârbu and N. Suri. Runtime behavior-based profiling of OS drivers. *http://www.deeds.informatik.tu-darmstadt.de/research/TR/TR-TUD-DEEDS-05-02-2007-Sarbu.pdf*, TR-TUD-DEEDS-05-02-2007, 2007.

[21] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. on Computer Systems*, 23(1):77–110, 2005.

[22] E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15, Issue: 5:54–59, 1998.

[23] E. J. Weyuker. Using operational distributions to judge testing progress. In *ACM Symposium on Applied Computing*, pp. 1118–1122, 2003

[24] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. on Software Engineering*, 17, Issue: 7:703–711, 1991.