Empir Software Eng (2009) manuscript No.
(will be inserted by the editor)

Profiling the Operational Behavior of OS Device Drivers

Constantin Sârbu · Andréas Johansson · Neeraj Suri · Nachiappan Nagappan

Received: date / Accepted: date

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. Currently, DD testing entails a broad spectrum of techniques, where *static* (requiring source code) and *dynamic* (requiring the executable image) and *static-dynamic* testing combinations are employed. Despite the sustained and improving test efforts in the field of driver development, DDs still represent a significant cause of system outages as the coverage is invariably limited by test resources and release time considerations. The basic factor is the inability to exhaustively assess and then cover the operational states, leading to releases of inadequately tested DDs. Consequently, 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 and help focus the test efforts.

Focusing on characterizing DD operational activities while assuming no access to source code, 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 XP and Vista drivers using various performance and stability benchmarks as workloads substantiates our proposed approach.

Keywords Operating System · Device Driver · Runtime Behavior · Operational Profile · Empirical Studies

1 Introduction

Currently, the largest (and also the most evolving) part of an OS is its interface to the hardware devices, as represented by the *device drivers*. Recent research (Albinet et al 2004; Arlat et al 2002; Duraes and Madeira 2003; Ganapathi et al 2006; Simpson 2003; Swift et al 2005) has shown that unfortunately DDs constitute a dominant cause of OS failures. Under a feature-driven market pressure, DDs are often released without exhaustive testing,

Constantin Sârbu (✉) · Neeraj Suri
Technische Universität Darmstadt
Darmstadt, Germany
E-mail: cs@cs.tu-darmstadt.de

Andréas Johansson
Volvo Technology Corporation
Gothenburg, Sweden
E-mail: andreas.olof.johansson@volvo.com

Neeraj Suri
E-mail: suri@cs.tu-darmstadt.de

Nachiappan Nagappan
Microsoft Research
Redmond, WA, USA
E-mail: nachin@microsoft.com

usually exhibiting a higher defect density compared to the more mature OS kernel (Chou et al 2001). Moreover, DDs are typically delivered as binaries constraining potential testing campaigns to black-box strategies. In addition, the set of loaded drivers used for testing is likely different than in deployed systems. Consequently, the coverage obtained from the limited test configurations may not accurately match the wider spectrum of deployed system configurations.

On this basis an important consideration is the DD testing under "field" conditions. While multiple sophisticated static testing techniques for DDs exist (Ball et al 2004; Mendonca and Neves 2007; Nagappan et al 2005), 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 functionalities actually executed, their sequence and occurrence patterns. Using this valuable information, we believe that subsequent test campaigns can primarily target code 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 (Weyuker 1998), 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 DD testing, our approach is based on monitoring the interface between the OS kernel and the DDs. At this interface, I/O traffic is captured and analyzed to build a state model of the DD. The state of a DD is represented by the set of DD functionalities observed to be in execution at specified instants. The transitions between states are triggered by incoming and outgoing (i.e., from a DD's perspective) I/O requests. The resulting behavioral model is used to discover execution hotspots in terms of frequently visited states of the DD 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. As an effort to validate the presented theoretical aspects, an empirical evaluation of actual Windows XP and Vista DDs is presented as case study.

Overall, the work in this paper focuses on providing a general operational state profiling framework for DDs. In addition, our work is an effort towards an improved DD test paradigm and not a test method *per se* (developing a comprehensive stand alone testing framework is not the intent of the current paper though is a subject of our ongoing research).

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 approach is presented in Section 5, along with detailed case studies in Section 6. Section 7 discusses the experimental issues and Section 8 provides a discussion on the overall findings.

2 Related Work

Software-Implemented Fault Injection (SWIFI) is a technique widely used to assert the robustness of black-box DDs (Duraes and Madeira 2003; Johansson and Suri 2005; Mendonca and Neves 2007). In industry, the main OS developers periodically release improved specifications and tools to minimize the risk of launching faulty DDs (Ball et al 2004) and make efforts towards determining the "requisite" amount of testing (Oney 2003; Nagappan et al 2005). Unfortunately, in spite of considerable advancements in DD testing (Albinet et al 2004; Arlat et al 2002; Chou et al 2001; Swift et al 2005; Ganapathi et al 2006; Duraes and Madeira 2003), DDs are still a prominent cause of system failures, as generally, test case selection and the code coverage thereof is problematic.

Musa's work (Musa 2004) on reliability engineering suggests that the overall testing economy can be improved by prioritizing testing activities on specific functionalities with higher impact on the component's runtime operation. Similarly, Weyuker (Weyuker 2003; Weyuker and Jeng 1991) recommends focusing on testing functionalities

with high occurrence probabilities. Results from the area of software defect localization show that faults tend to cluster in the OS code (Möller and Paulish 1993; Chou et al 2001). Thus, a strategy aimed at clustering the code into functionality-related parts and then testing based on their operational occurrence is desirable, especially when the resources allocated for testing are limited. Weyuker (Weyuker 1998) underlines the necessity to test COTS components in their new operational environments even though they were tested by their developers or third-party testers.

Rothermel et al. (Rothermel et al 2001) empirically examined several verification techniques showing that prioritization can substantially improve fault detection. Specifically, they assert 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 work 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 (McMaster and Memon 2005) introduced a statistical method for test effort reduction based on the call stacks recorded at runtime. While their approach effectively captures the dynamic program behavior, it is specific to single-threaded, user-space programs, though not directly applicable to DDs (as they run in an arbitrary thread context). Leon and Podgurski (Leon and Podgurski 2003) evaluated diverse techniques for test-case filtering using cluster analysis on large populations of program profiles, highlighting them as a prerequisite for test effort reduction.

Avritzer and Larson (Avritzer and Larson 1993) proposed an approach to describe the load of a large telecom system. They introduce a load testing technique called Deterministic Markov State Testing for describing the operational model of a telecommunication system. The incoming and completion of five types of telephone calls define the state of the system. However, as the work was driven by the necessity to test the given telecommunication system, knowledge of the system internals was intensively used, thus limiting the usability of the method to other, specific system. In contrast to their method, our approach is generalized to OS add-on components and to DDs.

While the proportion of lines of kernel code corresponding to drivers is on the rise mostly due to higher OS support for peripherals (Swift et al 2005), the need for novel driver-specific testing approaches and tools is also increasing. As a consequence, various paradigms were successfully applied to testing OS code and in particular to DDs. Notable runtime testing tools include Rational's `PURIFY` (Rational 2008) and Microsoft's `Driver Verifier` (Microsoft Corp. 2008). Such tools instrument a binary image with a set of checks, thus enabling the examination of the respective binary's runtime behavior. The checks are designed to detect illegal activity of the instrumented binary, avoiding potential system corruption. Specifically designed for Windows DDs, `Driver Verifier` is included in XP and Vista releases and continually enhanced with each new version of the OS. The major weakness of the runtime instrumentation tools resides in the dependency on the quality of the checks (i.e., if a condition goes unchecked, no corruption is signaled).

Using a different paradigm, multiple other research efforts target source code to detect errors at compile time, thus aiming at a better coverage of the DD than that offered by the binary instrumentation methods. For instance, this approach is used at Microsoft by the `SDV/SLAM` tool (Ball et al 2006) which checks if an "abstracted" (i.e., simplified) version of the DDs follow the rules stated by the OS kernel's API. The "abstract DD" is run inside a "hostile" execution environment, and a symbolic model checker is used to validate its activity against a set of rules. The abstracted image of the DD is used instead of the real image in order to keep the number of DD states within manageable limits for the model checker, this inducing a limitation of the coverage for the real DD.

To protect faulty DDs affecting the rest of the OS, another paradigm assumes running the DD in isolation. Examples include tools like Nooks (Swift et al 2002) or SafeDrive (Zhou et al 2006). This approach supposes that the execution of the real driver (or a slightly modified version thereof) inside a virtual execution environment. All execution traces of the targeted DD are monitored to prevent potential failures thereof to propagate to the rest of the OS, but this comes at a high performance overhead.

As a preliminary basis for the concepts presented in this paper, in our previous work (Sârbu et al 2006) we introduced a monitoring approach for state identification considering the functionality currently serviced, as seen from the OS – driver interface. In a parallel effort to characterize the operational behavior of DDs (Sârbu et al 2008), we identified the DD parts which are activated at runtime by using a methodology that permits (a) identification of the code paths followed by the DD in the operational mode and (b) clustering thereof in terms of relative sequence similarity. In a pre-testing phase, the parameters of the resulted clustering were analyzed and tuned to reveal the minimal subset of code paths that must be taken by a subsequent test campaign. Under the assumption that the test effort is equally distributed among the followed code paths (i.e., testing any two code paths is equally hard), we

showed that by covering only a subset of execution paths the testing effort can substantially be reduced without sacrificing adequacy.

3 Driver State Definition via I/O Traffic

We now introduce our system model and the background behind the state model for DDs. Fig. 1 represents a typical computer system equipped with a COTS OS supporting a set of applications (the *system workload*) using services provided by the OS.

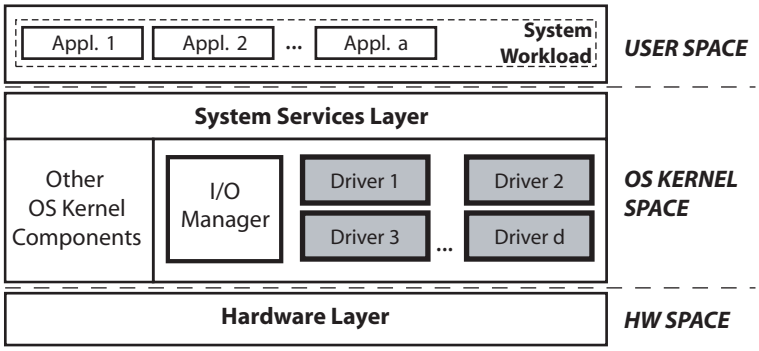


Fig. 1 The considered system model

This paper focuses on the communication interface between the *I/O Manager* and the *DDs* 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 *DD* is an independent OS component handling the communication with one or more equivalent peripherals.

For instance, a user application reads the content of a file stored on the local hard-disk. A `read` request is issued and passed to the OS by means of standardized functions defined by the System Services layer. Next, the request is handled by the I/O Manager which transforms it into a command for the driver associated with the local hard-disk. The driver accesses the hard-disk controller, and the controller instructs the disk drive's heads to move to the proper position and start reading data. The disk driver stores read data in a buffer which is made available to the application that initially issued the `read` request.

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 flow between the I/O Manager and the DDs is analyzed to characterize the activity of a DD. At this level Windows uses a shared memory communication scheme as specified by the Windows Driver Model (WDM) (Oney 2003).

According to WDM, the I/O Manager builds a request data structure and populates it with the parameters necessary for the driver to start resolving the request. Next, the I/O Manager informs the responsible driver that a request is available. When the driver finishes executing the code associated with resolving the request, it fills the result fields of the structure and passes it back to the I/O Manager. The I/O Manager unpacks the data structure and forwards the results to the user space application that requested the I/O. The data structure used for passing the request between the I/O Manager and drivers are called *I/O Request Packets* (IRP). The current WDM specifications define 28 IRP types, each of them associated with a certain operation supported by the driver (e.g., CREATE, READ, WRITE, CLOSE, etc.).

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 type of IRP. The set of dispatch routines supported by the driver can be easily discovered by examining the OS kernel structure describing the driver, i.e., the DRIVER_OBJECT. A standard method (*DriverInit*) of the driver executes after the driver has been loaded into the kernel. Among other actions, *DriverInit* fills a field of the DRIVER_OBJECT structure with pointers to the supported dispatch routines. Hence, simply by inspecting the DRIVER_OBJECT structure the set of supported IRP types can be identified.

3.1 The State of a Driver

From a testing perspective, the ability to precisely pinpoint which functionality the system-under-test executes at any specific instant is of critical importance as a basis for observability. This is not a trivial task when testing OSs (or components thereof) as they are complex and dynamic, entailing a high level of non-determinism and often being delivered without source code. These constraints constantly challenge the software testing community to investigate new methods to define and accurately capture the *state* of such a system.

3.2 Driver Modes and Transitions

The state of a DD is characterized by the handled IRP requests. As we assume no access to the driver's source code, we are constrained to use a relaxed definition of *state* to accommodate only the available information (i.e., the observable communication at the interface of the driver). Onwards, we define this relaxed state as the "driver *mode*". Hence, a DD is idle from the initialization instant until the first IRP request is received. The DD is in the "*processing* IRP_i" state from the instant when IRP_i is received and until the DD announces its completion. In each mode, the DD executes a dispatch routine specific to the type of the received IRP, as specified by the WDM (Oney 2003).

Some IRPs are processed concurrently by the DD, i.e., processing IRP_j can start before IRP_i is completed (assuming that IRP_i was initiated before IRP_j but its activity is not finished yet). 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 (Sârbu et al 2006) we defined the *mode* of a driver D as follows:

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

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

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

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 is 2^n.

Definition 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 (Sârbu et al 2006) 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.

To illustrate the presented concepts of modes, transitions and operational profile of a DD let us consider an example. Fig. 2 depicts the state space of a hypothetical driver supporting four distinct IRPs, i.e., CREATE, READ, WRITE and CLOSE. The leftmost bit is set while the driver performs the functionality associated with CREATE, the second leftmost bit is set while the driver performs READ and so forth. Note that the driver can execute several activities of different types concurrently, in which case the binary string contains several bits set. In Fig. 2 we shade the subset of modes visited by the driver for a hypothetical workload.

As implied by the definition of driver mode, transitions between modes are triggered by receiving and completing I/O requests. As the I/O Manager serializes both the sending and receival of IRPs, only one bit can change 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 ($T_{OP} \ll n \cdot 2^n$) (Sârbu et al 2006), the actual state space that needs consideration is significantly reduced. If needed, such a DD's OP can also be used to pro-actively discover new modes by forcing the DD to traverse non-transited edges (e.g., for robustness testing).

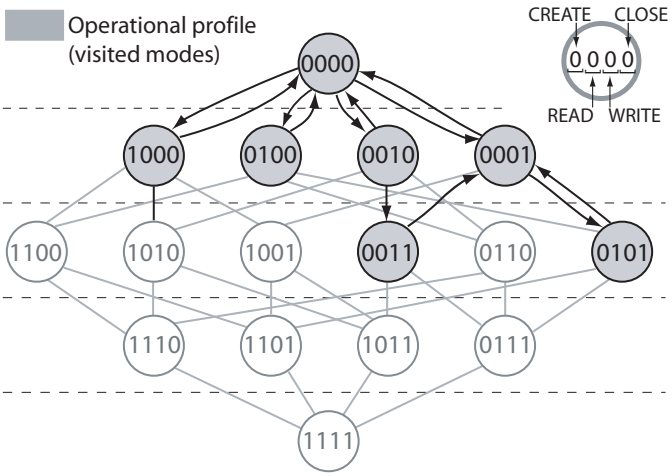


Fig. 2 An example of state space and OP for a driver supporting four distinct IRP types

3.3 Characterizing a Driver's Behavior

In our prior work we experimentally identified the OP of the serial port DD provided with Windows XP SP2 (Sârbu et al 2006). 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 DDs activity as the ability to distinguish among the visited modes and transitions is missing.

In the next section we enhance the captured OPs by introducing additional metrics for an accurate characterization of the DD'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 DD's code. This assumption is justified for testing black-box device drivers as the information about their runtime behavior is implicitly limited. For instance, when information about the runtime behavior of the driver permits prioritization, a robustness testing strategy can first target the modes and transitions which were least visited, under the assumption that the functionalities utilized more commonly were already tested by the driver's developers. Similarly, under the same assumption, the modes where the driver is concurrently executing several functionalities triggered by IRPs are also good candidates for robustness testing.

4 Quantifiers of Runtime Behavior

An accurate characterization of the operational behavior of a software (SW) component is desirable for establishing effective testing methods. Our interest focuses on COTS device drivers, SW components known for their limited observability at runtime. As test completeness is hard to reach for drivers mainly due to their complexity, testers usually choose to primarily test the key functionalities (Mendonca and Neves 2007). Even when it is assisted by tools with certain degrees of automation, this selection remains a process based on the tester's subjective experience in prioritization.

Hence, this section presents a set of quantifiers developed for differentiating among the visited modes and transitions of a DD's OP. Using them, the relative frequencies of the visited modes and transitions can be observed and analyzed, 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 DD's activity in the field can be used for workload assessments, usage/failure data collection or post-mortem debugging. Moreover, different workloads can be statistically compared to reveal the DD modes with higher probability of being reached in the field.

From a testing perspective, the metrics associated with the DD modes indicate *how often* and *how much time is spent* in each mode. Both represent valuable information about the operational mode of a DD, guiding the

subsequent test campaigns through enabling inter-mode priority rankings. These rankings are tunable to the main purpose of the test activity. For instance, if the goal is early discovery of the defects with high probability to occur in the field, then the test campaign should start by first covering the mostly visited DD mode, and continue in the decreasing order of the sojourn rate of the remaining modes until all of them are covered or the resources allocated for testing are depleted.

In this paper we also introduce metrics for transitions among the modes belonging to an OP. While we believe that the need for mode quantifiers for testing is intuitive (modes are abstract representations for the DD code), the arguments supporting the development of transition quantifiers are easier understood via a simple example. For instance, assuming that a tester wants to test the mode `0011` in Fig. 2. To "drive" the DD into that mode, one needs to design a test case which calls a `WRITE` followed by a `CLOSE` I/O call. A simple issue of the two commands in this sequence might not be sufficient as the DD might finish the `WRITE` operation before the `CLOSE` is called. If this happens, the test is applied to the current mode `0001` (instead of the targeted `0011`!). Please note that the test parameters alone cannot guarantee that the desired mode is reached, a decisive timing aspect is also involved. Our transition quantifiers probabilistically capture this aspect, enabling the development of complex test cases (i.e., sequences of I/O calls instead of singletons). Hence, they permit computing the probability to reach the mode of interest depending on the current mode. Specifically, after each hop in a sequence of I/O calls, this probability is re-calculated in terms of the current mode.

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 deviations from the expected behavior of a DD by observing a population of runs of the selected workload and finding the OP which diverges from the rest of the runs in terms of one (or multiple) quantifiers. Section 6.3 illustrates the workload comparison procedure enabled by our approach.

For testing purposes, our metrics can be used to quantitatively compare the effectiveness of test cases (or test suites) on the driver-under-test. For instance, if the DD code is not available, 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 by removing the redundant test cases. Section 6.4 illustrates of the test space reduction of our method. We believe that access to the DD source code might reveal additional information about the relation between the mode coverage and code coverage and our current research interests include an extensive study of this relation. Also, Table 5 in Section 6 contains guidelines for computing, using and interpreting the quantifiers, based on our empirical expertise accumulated in the process of collecting and analyzing operational behaviors of Windows DDs.

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 likelihood 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 an ordered pair 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 within a Hamming distance of 1 from each other.

Definition 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.

Definition 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*.

Definition 5 (Mode Occurrence Weight: MOW_i) The occurrence weight of mode $M_i^D \in M$ represents a quantification of a driver's likelihood 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 OPs for building reliable SW components proposed by Musa (Musa 2004). In contrast to (Musa 2004), 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 quantifier indicates the percentage of mode sojourns that put the DD in an idle state, i.e., waiting for IRP requests.

Similar 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.

Definition 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 likelihood 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.

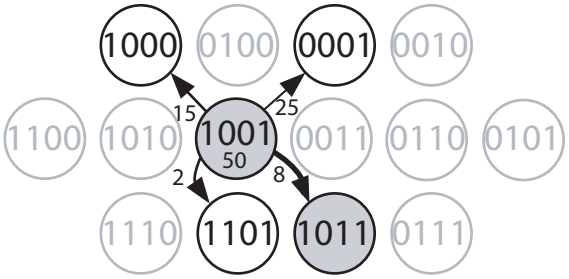


Fig. 3 Calculating $TOW_{t_{1001,1011}}$

For instance, consider the situation depicted in Fig. 3, 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 (i.e., 16% probability that the mode `1011` will be visited next when the mode `1001` is current).

4.2 A Time-based Quantifier for Modes

To increase the accuracy of DD profiling, both spatial and temporal dimensions need to be considered. We regard the duration of a DD's activity not just an artifact of the device's inherent slowness but as an 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 (Musa and Okumoto 1984)). 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 metric for edges is superfluous.

The overall time spent by the DD in each mode reveals valuable information about the latencies of various IRP-related activities of a DD. 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 "slow" devices as disks 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.

Definition 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 or in combination. In order to facilitate combinations of the two metrics, we propose a compound quantifier capturing these dimensions of the profiled DD's activity, namely the *mode compound weight*.

Definition 8 (Mode Compound Weight: MCW_i) The compound weight of a mode $M_i^D \in M$ is given by the expression (where $\lambda \in \mathbb{R}, 0 \leq \lambda \leq 1$):

$$MCW_i(\lambda) = \lambda MOW_i + (1 - \lambda)MTW_i \tag{4}$$

By varying λ, this quantifier can be biased towards either occurrence or temporal dimension as needed for the test requirements. 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 OP quantifiers be used to compare the effects of different workloads on DDs?

Q2: What is the magnitude of test space reduction introduced by the usage of our OP quantifiers?

To answer these questions, the following subsections show how the rankings based on execution quantifiers are obtained (Section 6.2), how workloads are compared among each other (Sections 6.3) and how are our OP quantifiers usable for test space reduction (Section 6.4).

5.1 Experimental Setup and Analysis Strategy

To capture the IRP flow, we have built a lightweight "*filter driver*" interposed between the I/O Manager and a DD of our choice (Fig. 4). 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 traffic 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 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

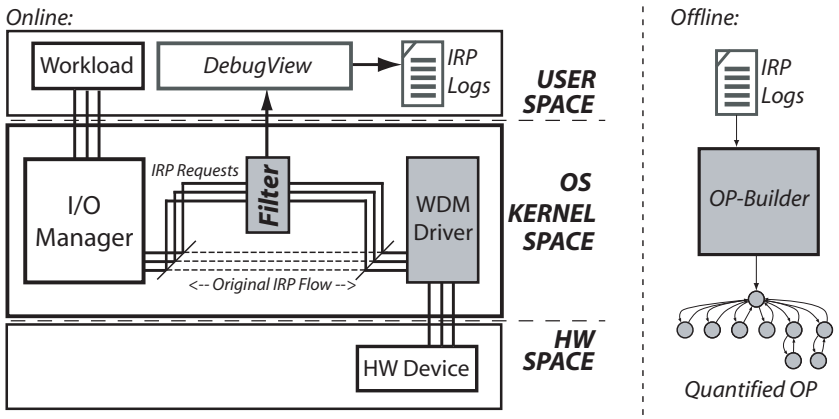


Fig. 4 The experimental setup; the online (monitoring) and offline (analyzing) phases

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 behaviors were investigated in this paper.

To compute the OPs for different DDs, we have designed a tool (*OP-Builder*) that processes the logs and outputs the DD's OP, together with all runtime quantifiers. The figures 26–29 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 (Russinovich 2008).

5.2 Studied Drivers and Workloads

To put our work in context with Windows DDs, we first describe the different driver types as specified by WDM (Oney 2003). WDM uses a layered architecture of drivers similar to the one depicted in Fig. 5. Each hardware device has at least two associated DDs, the *function driver* and the *bus driver*. The former handles most of the work being done in order to manage the associated peripheral properly, while the latter manages the communication bus which connects the peripheral to the rest of the computing system (i.e., ISA, PCI, USB etc.). Some devices have additional layers, consisting of *filter drivers* which wrap the function driver. A filter driver is responsible for modifying the behavior of the main function driver and it can be located either above or below it. WDM does not limit the number of filter drivers a hardware device can have. This mechanism permits incremental changes to the main behavior of the function driver, thus enabling modifications that would otherwise require source code access.

Additionally, the function drivers come in two flavors, depending on their implementation, as *monolithic* and as combinations of a *class driver* and a *minidriver*. A monolithic driver encapsulates all of the functionality needed to support a hardware device. The same functionality can also be modularly implemented as a combination of a class- and a mini-driver. In this case, the class driver manages the entire generic class of devices, while the minidriver handles only the vendor-specific functional characteristics of a certain device. Usually, the class drivers are provided by Microsoft and writing the minidriver is generally the hardware manufacturer's task.

As the case studies presented in this paper are valid for WDM-compliant drivers, they are readily transferrable to Vista drivers, too. For its latest commercial operating system (Vista) Microsoft introduced the Windows Driver Foundation (WDF) (Orwick and Smith 2007). WDF defines two main driver categories, the user-mode drivers (User-Mode Driver Framework - UMDF) and the kernel-mode drivers (Kernel-Mode Driver Framework - KMDF). As KMDF represent an extension of the earlier WDM, our case studies on WDM drivers hold also for Vista's KMDF-compliant drivers.

Currently, most of the drivers available for Windows operating systems are belonging to the WDM and KMDF classes, with the balance of the sheer number of DDs slowly moving from WDM towards KMDF drivers as Vista's popularity is increasing. The error reporting facility of Windows Vista enabled Microsoft's researchers to estimate the unique devices attached to Vista systems to 390000, while the DD population is increasing every day with 25 new and 100 revised DDs on average (Orgovan 2008).

Currently, the DDs onto which our approach cannot be directly applied are the drivers for display and video capture adapters, printers, scanners and SCSI storage adapters, but these represent very few cases reported to the total universe of Windows DDs. The vast majority of Windows drivers is still represented by WDM/KMDF DDs (Microsoft 2006), emphasizing the applicability of our profiling methodology presented in this paper.

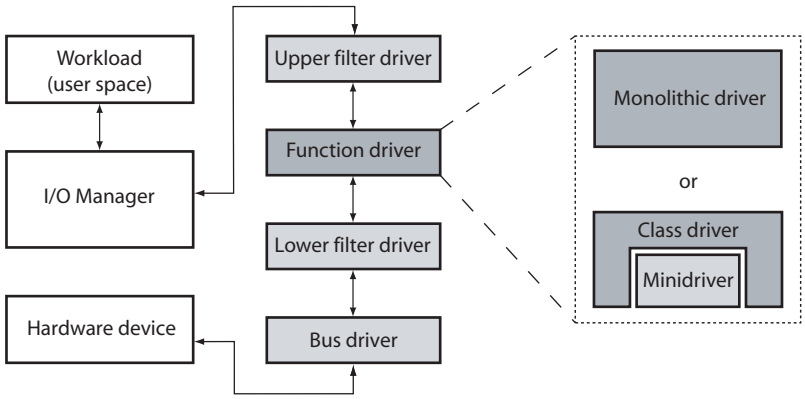


Fig. 5 Driver layering in WDM. Arrows represent the I/O traffic between a user-space application (*workload*) performing I/O and an associated *hardware device*. The function driver can be implemented as either a *monolithic driver* or as a *class driver - minidriver* combination.

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` – both for XP and Vista), an ethernet card driver (`sisnic.sys`), a floppy driver (`flpydisk.sys` – both for XP and Vista) and a parallel port driver (`parport.sys`). All DDs are provided (and digitally signed) by Microsoft, except the `sisnic.sys`, which is provided by the SiS Corporation.

The drivers selected for the experimental evaluation presented in this paper span all the different driver types described earlier. Table 1 lists their main features.

Table 1 The studied device drivers and their characteristics. The first column contains the short name used onwards to refer to the respective DDs, differentiating among DDs profiled under the Windows XP or Vista OSs.

Short	Executable Image (Version)	Managed Device	Used as	Features
cdrom_XP	cdrom.sys (5.1.2600.2180)	DVD drive	Class driver, provides access to CD ROMs and DVD ROMs	PnP, power management and media change notification (autorun)
cdrom_Vista	cdrom.sys (6.0.6000.16386)			
floppy_XP	flpydisk.sys (5.1.2600.2180)	Floppy drive	Block-device, legacy driver (monolithic)	Sits on top of the floppy disk controller in the driver stack mediating the communication with the user-level application which calls into the floppy disk controller
floppy_Vista	flpydisk.sys (6.0.6000.16386)			
parallel_XP	parport.sys (5.1.2600.2180)	Parallel port	Parallel port function driver and parallel port bus driver	PnP, power management, WMI, raw access to all parallel devices. Can share the access to all parallel ports on the system and detects all parallel enumerable devices connected to the port
serial_XP	serial.sys (5.1.2600.2180)	Serial port	Function driver for legacy PnP COM ports or as a lower-level filter driver for PnP devices requiring 16550 UART interface	PnP, power management, WMI. Controls interrupts and communication with device hardware (monolithic). Used in conjunction with `serenum.sys` (which acts as a device upper filter for `serial.sys`
ethernet_XP	sisnic.sys (1.16.00.05)	Ethernet card	*No information available (distributed as binary only by the SiS Corp.)*	

Table 2 The workloads utilized to exercise the DDs and their experimental attributes, in terms of generated I/O traffic and operational profile size. The OPs captured for all the benchmark–driver–OS combinations listed in this table are presented in detail in Section 6.1.

Short	Benchmark	Driver Short	IRPs		Visited		Description / Duration [min:sec]
			Issued	Types	Modes	Edges	
C1	BurnInTest-Audio	**cdrom_XP**	1336	3 out of 9	4	6	Audio CD test mode [07:40]
C2	BurnInTest-Data		71012	2 out of 9	3	4	Data CD read and verify mode [01:03]
C3	BurnInTest-Audio	**cdrom_Vista**	2170	5 out of 9	6	10	Audio CD test mode [03:50]
C4	BurnInTest-Data		51034	5 out of 9	6	10	Data CD read and verify mode [01:02]
F1	BurnInTest	**floppy_XP**	2946	5 out of 6	6	10	Various pattern read and write [05:03]
F2	Sandra Benchmark		19598	5 out of 6	9	16	Perf. benchmark various filesize [23:40]
F3	DC2		50396	4 out of 6	5	8	MS Device Path Exerciser [00:32]
F4	DevMgr-Disable		10	1 out of 6	2	2	Device Manager – disable drive [00:0.01]
F5	DevMgr-Enable		400	3 out of 6	4	6	Device Manager – enable drive [00:17]
F6	F1 – F5, sequentially		63396	5 out of 6	6	10	Sequential execution of F1–F5 [31:00]
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]
F9	BurnInTest	**floppy_Vista**	3008	6 out of 6	7	12	Various pattern read and write [05:03]
F10	DevMgr-Disable		10	3 out of 6	4	6	Device Manager – disable drive [00:.004]
F11	DevMgr-Enable		245	6 out of 6	7	12	Device Manager – enable drive [00:0.03]
P1	DC2	**parallel_XP**	48530	6 out of 6	7	12	MS Device Path Exerciser [00:5.7]
S1	BurnInTest	**serial_XP**	11568	6 out of 6	9	16	COM1 loop-back test [05:00]
E1	BurnInTest	**ethernet_XP**	2480	4 out of 28	5	8	TCP/UDP full duplex [05:03]
E2	DevMgr-Disable		6	1 out of 28	2	2	Device Manager–dis. Ethernet [00:01]
E3	DevMgr-Enable		114	6 out of 28	7	12	Device Manager–enable Ethernet [00:02]

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, constructed the OP graphs as in Fig. 26 and calculated the OP quantifiers defined in Section 4.

Besides commercial benchmarks testing the performance and reliability of the peripherals under various conditions, it is worth mentioning that we have 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 2. In the full spectrum of our experiments all the other mentioned DDs also showed the effectiveness of our OP profiling method. The detailed results are presented in Section 6.1.

6 Data and Analysis

6.1 Detailed Experimental Results

In this section we present the detailed results of the experiments listed in Table 2, in terms of the obtained OPs. All the figures in this section (Fig. 6–25) are structurally similar, the information contained in the nodes of the OP graphs is (see mode `000` in Fig. 6, from top to bottom): (a) Mode name (e.g., `000`); (b) MCW value (e.g., `0.5008`) and (c) MTW value (e.g., `0.9951`). Similarly, the transitions are marked with the TOW weight (see transition `000` → `001` in Fig. 6).

6.1.1 Operational Profiles of the `cdrom_XP` Driver

For the workload C1 (Fig. 6) 1336 IRPs were generated, belonging to 3 types (CREATE, CLOSE and DEVICE_CONTROL) out of a total of 9 types supported by the `cdrom_XP` driver. The most accessed driver operation was CREATE, the other two were only called 8 times each.

Figure 7 illustrates the behavior of the `cdrom_XP` driver under the workload generated by the BurnInTest application by testing a data CD media. 71012 IRPs were generated in total, belonging to 2 types (READ and

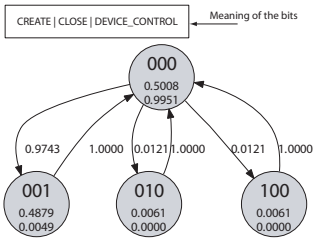


Fig. 6 `cdrom_XP` profile for the workload C1: BurnInTest - Audio

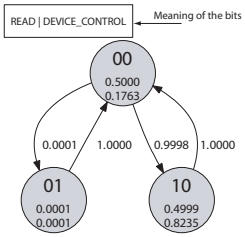


Fig. 7 `cdrom_XP` profile for the workload C2: BurnInTest - Data

DEVICE_CONTROL). READ was the most accessed driver operation under this workload, the driver spending here 82.35% of the total experiment time (1 minute and 3 seconds).

6.1.2 Operational Profiles of the `cdrom_Vista` Driver

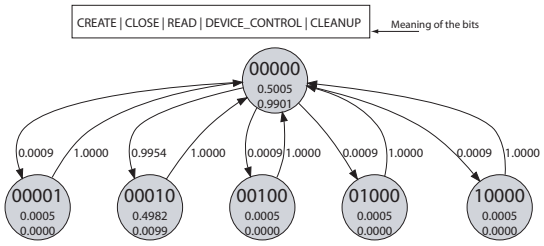


Fig. 8 `cdrom_Vista` profile for the workload C3: BurnInTest - Audio

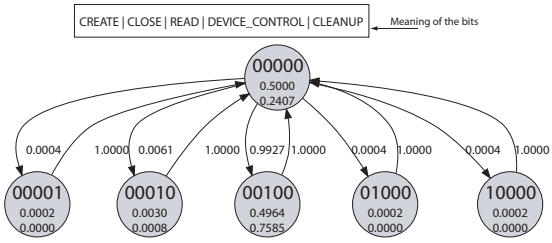


Fig. 9 `cdrom_Vista` profile for the workload C4: BurnInTest - Data

Under Vista, we re-used the same workloads that were used to exercise the cdrom.sys DD under Windows XP. Though, we observed large differences in both shapes and quantifiers of the obtained OPs from figures 6 and 7 versus the figures 8 and 9.

These differences can be explained by the fact that different I/O Manager units build the I/O traffic for the responsible DD, as we are dealing with different OSs. For instance, the `cdrom_Vista` DD received five types of IRPs from the Vista I/O Manager on behalf on both C3 and C4 workloads, while under Windows XP only three and respectively, two distinct types were issued (see Table 2). By comparing figures 6 and 8 in terms of issued IRPs, it becomes apparent that the `cdrom_Vista` DD received two distinct IRPs (READ and CLEANUP) more than the `cdrom_XP` DD. Figures 7 and 9 show the same trend, this time three distinct IRPs (CREATE, CLOSE and CLEANUP) were additionally issued for the `cdrom_Vista` DD but not for the `cdrom_XP` DD.

As BurnIn Test benchmark completed successfully both under Vista and XP, the differences in the captured OPs indicate that the internal structures of the two versions of the cdrom.sys DDs is also very different, although the same functionality is provided by both DDs.

6.1.3 Operational Profiles of the `ethernet_XP` Driver

We exercised the ethernet driver (`ethernet_XP`) with three workloads. First we have used BurnInTest benchmark (Fig. 10), followed by disabling the driver from the Windows Device Manager (Fig. 11) and then re-enabling it (Fig. 12). Therefore, figures 11 and 12 illustrate the activity performed by the driver at loading and unloading. This discloses the execution of the functionalities executed very seldomly (i.e., only at load and unload of the DD), but which are critical for the correct and dependable performance of the DD.

While the disable operation requires only the execution of the activity performed by a single IRP (CLOSE), the enable operation is more complicated, six IRPs are called, all of them in conjunction with the PNP activity. As both Enable and Disable operations require the PNP operation to be executed, it indicates the high importance this IRP type has for the management of the driver in the OS.

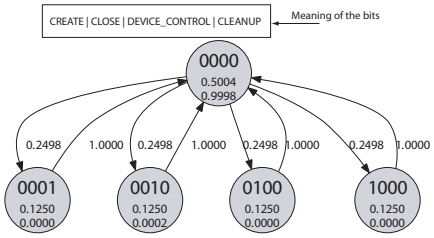


Fig. 10 `ethernet_XP` profile for the workload E1: BurnInTest

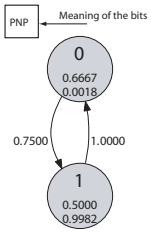


Fig. 11 `ethernet_XP` profile for the workload E2: Device Manager - Disable driver

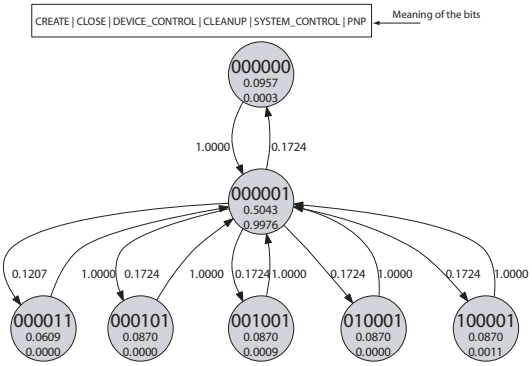


Fig. 12 `ethernet_XP` profile for the workload E3: Device Manager - Enable driver

6.1.4 Operational Profiles of the `floppy_XP` Driver

This section presents and discusses the OPs obtained for the `floppy_XP` DD by exercising it with the F1–F8 workloads, as described in Table 2. The same `floppy_XP` DD along with the mentioned workloads are also the subjects of a different study (further presented in Section 6.3) aimed at revealing the ability to compare workload effects using our OP quantifiers.

The OPs induced by the workloads F1 and F2 onto the `floppy_XP` driver are illustrated by the figures 13 and 14. The same two workloads were also selected to run concurrently, generating the F7 and F8 workloads.

BurnInTest accessed five IRP types without concurrent executions of the associated functionalities (Fig. 13) in the time interval spanning the experiment. In contrast, Sandra benchmark called exactly the same IRP types, but in a manner that put the driver in three additional modes located on the level 2 in the OP graph (Fig. 14 - the modes `00011`, `01010` and `10010`). In all these three modes, the floppy disk driver accessed CREATE, CLOSE and DEVICE_CONTROL in conjunction with the WRITE operation. Therefore, mode `00010` (WRITE) is both the most accessed and the mode where the driver spent most of the time.

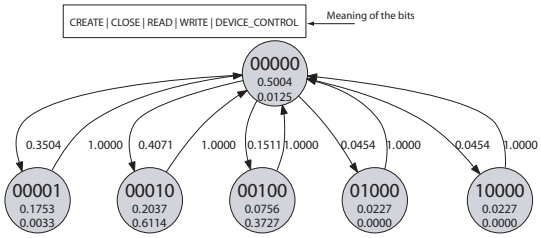


Fig. 13 `floppy_XP` profile for the workload F1: BurnInTest

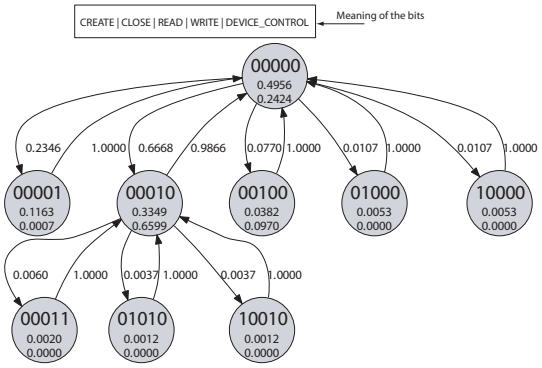


Fig. 14 `floppy_XP` profile for the workload F2: Sandra benchmark

However, the behavior of the `floppy_XP` DD was counter-intuitive. DC2 is a robustness test tool and given the fact that it generated a very large number of IRPs for the short interval of time it ran (50396 IRPs in 32 seconds), we had expected a large number of modes to be visited. Interestingly, the number of modes sojourned under DC2 (Fig. 15) was less than those for the workloads F1 and F2. Moreover, no modes associated with concurrent execution of driver functionality were visited. This might be an indication that the DC2 waits until the driver finishes the current test then resets the device settings in order to restart a new robustness test. This assumption is supported by the large number of times the mode responsible for DEVICE_CONTROL operations is accessed.

Interestingly, for the floppy disk driver both the enable and disable operations are using different IRPs than the ethernet driver. Instead issuing PNP requests for disabling it, the I/O Manager calls CLOSE to disable the `floppy_XP` driver (Fig. 16). Also, for enabling the driver (Fig. 17), only three IRP types are used (CREATE, CLOSE, DEVICE_CONTROL), an additional call to PNP is not required. We believe that these major differences, in the manner in which drivers are loaded and unloaded from the OS, originate in the functional and architectural variety among drivers following the WDM specifications.

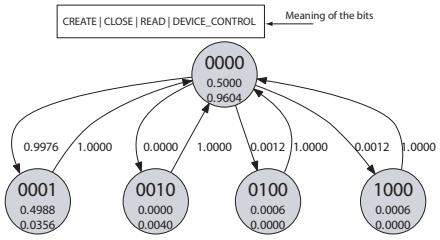


Fig. 15 `floppy_XP` profile for the workload F3: DC2 (Device Path Exerciser)

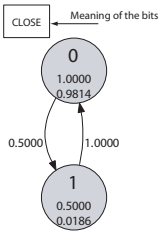


Fig. 16 `floppy_XP` profile for the workload F4: Device Manager - Disable driver

Figure 18 is a graphic representation of the OP when the workloads F1 to F5 were executed sequentially. This is an attempt to study the runtime behavior of the `floppy_XP` driver over longer periods of time, while the driver is subjected to multiple tasks. The driver executed mostly DEVICE_CONTROL operations in terms of number of sojourns to the respective mode, while WRITE was the most expensive operation in terms of time spent running the functionality associated with it.

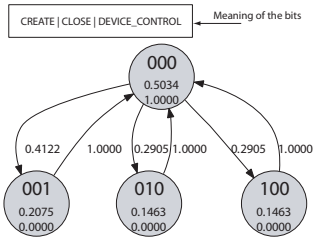


Fig. 17 `floppy_XP` profile for the workload F5: Device Manager - Enable driver

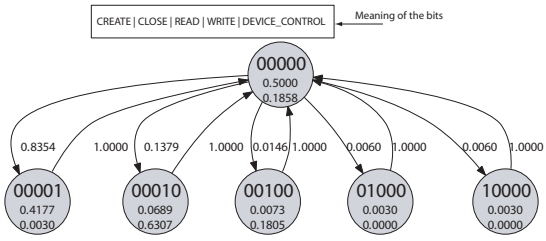


Fig. 18 `floppy_XP` profile for the workload F6: Sequential execution of F1–F5 workloads

The concurrent executions of F1 and F2 revealed that exactly the same modes and transitions were visited irrespective of the order in which the workloads were started. We now present the obtained OPs (Fig. 19 and 20). A detailed analysis and a quantitative comparison of these two workloads is presented in Section 6.3.

6.1.5 Operational Profiles of the `floppy_Vista` Driver

As for the cdrom.sys DD, we conducted a detailed profiling of the operational behavior also for the floppy disk DD, under Windows Vista and XP. The trend observed by comparing the two CDROM device drivers was confirmed by the comparison of the OPs of the floppy disk drivers installed under the two OSs. In general, under Vista more distinct IRP types are issued than under Windows XP, while the benchmarks produced the same results.

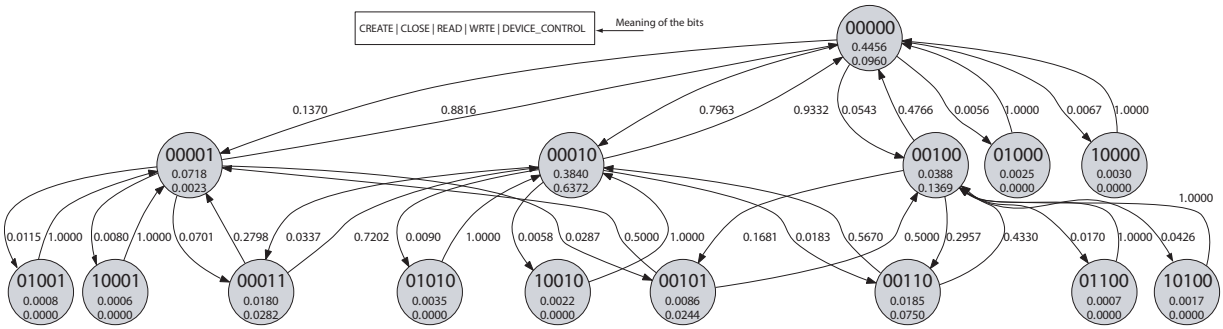


Fig. 19 `floppy_XP` profile for the workload F7: Concurrent execution of F1 and F2 (run I)

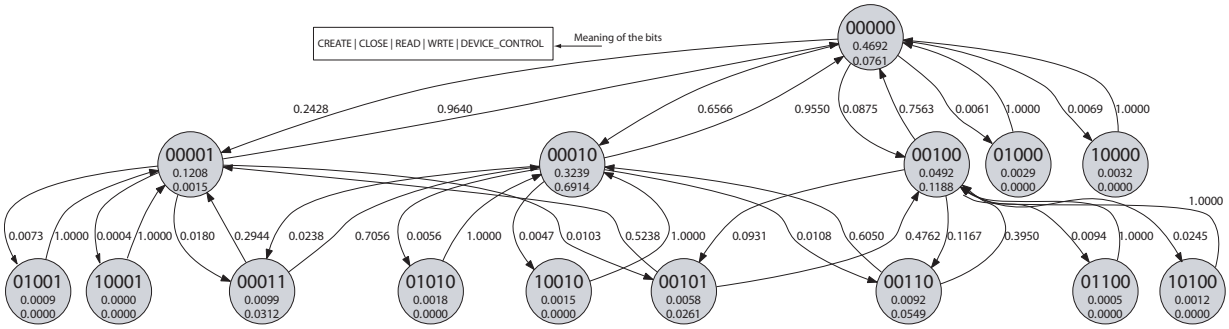


Fig. 20 `floppy_XP` profile for the workload F8: Concurrent execution of F1 and F2 (run II)

For the `floppy_Vista` DD, on behalf of the BurnInTest benchmark, the Vista's I/O Manager issued additionally to the XP's I/O Manager the CLEANUP IRP (see figures 13 and 21). The structure of the OP graph is similar in these two figures, only modes located on the first level were visited for both floppy disk DDs.

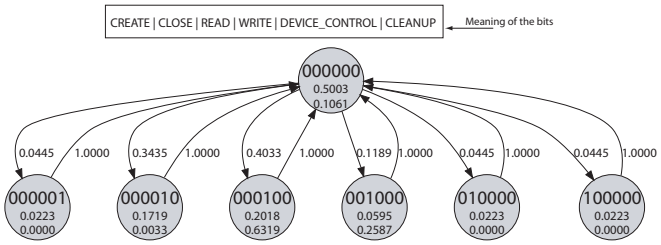


Fig. 21 `floppy_Vista` profile for the workload F9: BurnInTest

By comparing the OPs obtained for Disable operations (workloads F4 and F10, figures 16 and 22), we observed that two more distinct IRP types were issued for `floppy_Vista` than for `floppy_XP`, hinting at a more complex mechanism for unloading DDs in Vista than in Windows XP, even though exactly the same number of IRPs were issued under both OSs (see Table 2).

In contrast to the DD unload mechanism, by comparing the install routines of the two OSs (figures 17 and 23), we observed that Vista is much faster than XP (0.03 seconds for Vista versus 17 seconds for XP). The number of issued IRPs under XP was reduced in Vista to almost half, even though the distinct types they belong to contains three more in Vista (QUERY_INFORMATION, CLEANUP and PNP).

6.1.6 Operational Profiles of the `parallel_XP` Driver

Figure 24 illustrates the behavior of the `parallel_XP` under the effect of the DC2 robustness tool. Similar to the OP captured in Fig. 15 for the `floppy_XP` DD, DC2 only accessed modes located on the first level in the case of the `parallel_XP`, too. For the parallel port, DC2 additionally called the QUERY_INFORMATION, CLEANUP

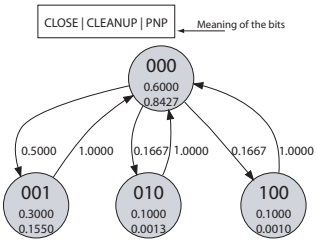


Fig. 22 `floppy_Vista` profile for the workload F10: Device Manager - Disable driver

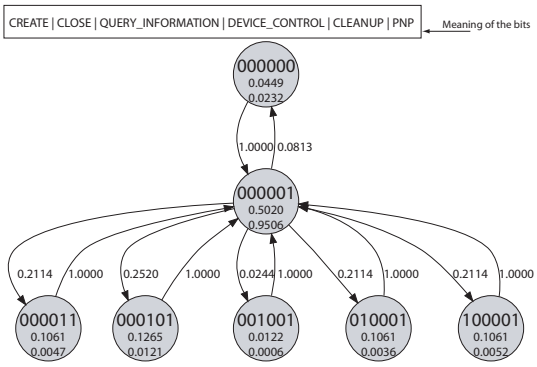


Fig. 23 `floppy_Vista` profile for the workload F11: Device Manager - Enable driver

and INTERNAL_DEVICE_CONTROL, but it did not call the READ IRP. Overall, the functionality associated with the DEVICE_CONTROL operation was mostly visited, indicating that the setting and getting of device capabilities represents an important activity from the perspective of the DC2 robustness test tool.

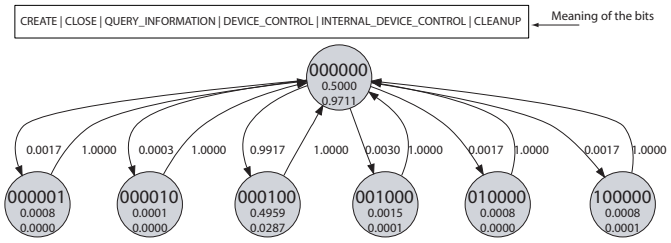


Fig. 24 `parallel_XP` profile for the workload P1: DC2 (Device Path Exerciser)

6.1.7 Operational Profiles of the `serial_XP` Driver

Figure 25 graphically represents the OP recorded for the `serial_XP` driver under the BurnInTest workload. Only two modes (`0100001` and `1000001`) were visited on the second level, most of the mode sojourns being made to the modes associated with READ (`0010000`) and WRITE (`0001000`) operations. The most of execution time was also spent in the WRITE mode.

6.2 A Prioritization Case Study: the `floppy_XP` Device Driver

Figure 26 depicts the OP of the `floppy_XP` driver, as exercised by the F2 workload (see Table 2). 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 Fig. 26 the darker gray hemispheres represent higher MOW or MTW values, revealing the frequently executed DD functionalities and the ones with longer execution times, respectively. The mode `000000` was predominantly visited 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 Fig. 26 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

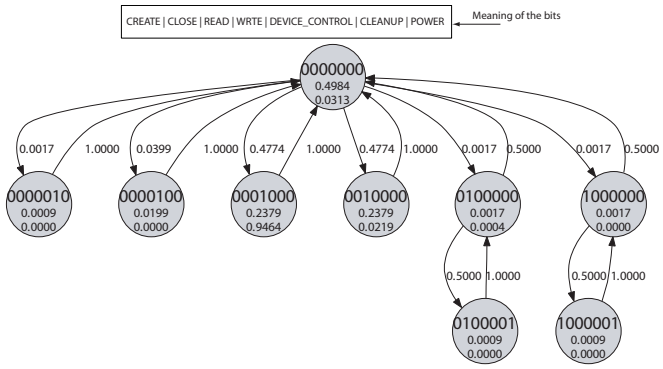
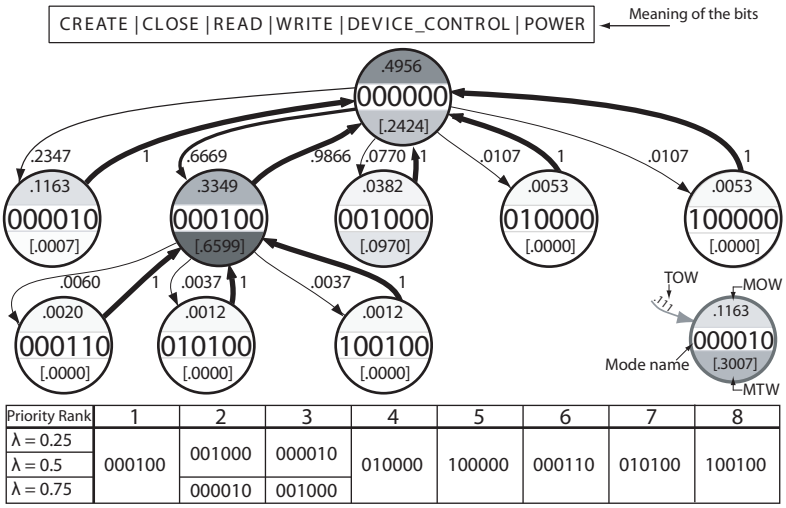


Fig. 25 `serial_XP` profile for the workload S1: BurnInTest



Priority Rank	1	2	3	4	5	6	7	8
λ = 0.25	000100	001000	000010	010000	100000	000110	010100	100100
λ = 0.5								
λ = 0.75		000010	001000					

Fig. 26 The OP for F2 and three prioritization cases; the figure is content-wise similar to Fig. 14, with the difference that here we visually emphasized the execution hotspots by using darker shades and thicker lines

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).

Fig. 27 illustrates the temporal evolution of F2 with different time windows. The upper part of Fig. 27 depicts driver evolution over the entire experiment duration (23 minutes and 40 seconds), where dots indicate mode sojourns. This representation reveals distinct execution patterns. For instance, the modes `000110`, `010100`, `100100` are visited only seldomly and also follow three times the same sojourn pattern. For better insight, the lower part of the figure shows the mode sojourn pattern in a selected 60μs time frame. This representation exposes the functionalities in execution at any instant enabling testers to reproduce the IRP sequence that brings the driver in a mode of interest (i.e., to create effective test cases).

6.3 Comparing Driver Runtime Profiles

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 OPs can be performed. Ideally, a minimal difference ensures post-release test adequacy (Weyuker 1998), though measuring it is not trivial without 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.

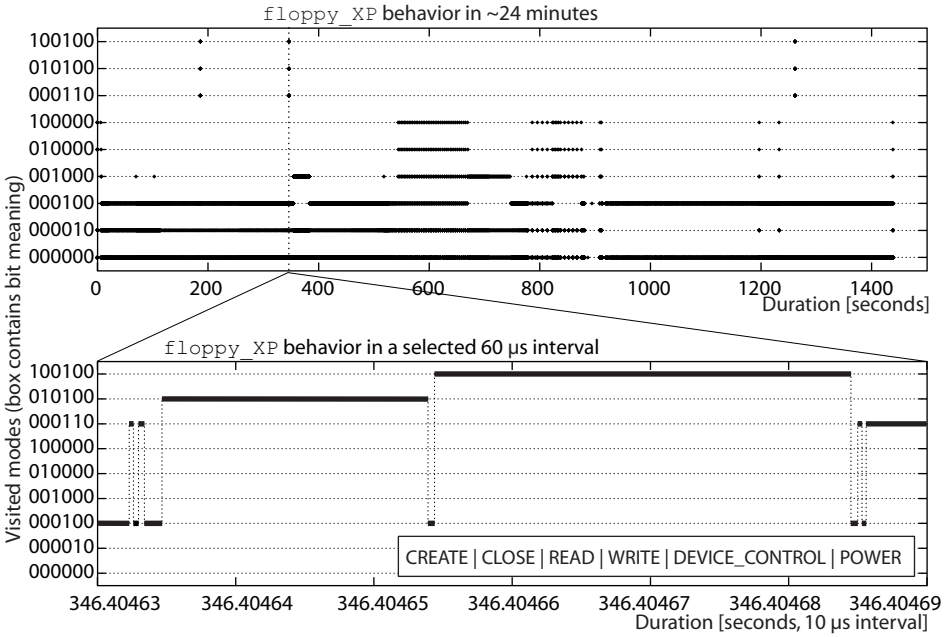


Fig. 27 Mode sojourn pattern for F2. Note that the DD is executing only in a single mode at any instant even if the upper part of the figure might suggest the opposite; this fact is more visible in the lower figure, where the sojourn pattern of a very short interval of time is presented, with transitions among modes depicted as vertical dotted lines.

We present a comparative study of two workloads for the floppy disk driver, workloads F7 and F8 (see Table 2). For F7, we first started the F2 workload, followed after a few minutes by 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 completed 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 figures 19 and 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 side-by-side comparison.

Fig. 28 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, Fig. 29 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.

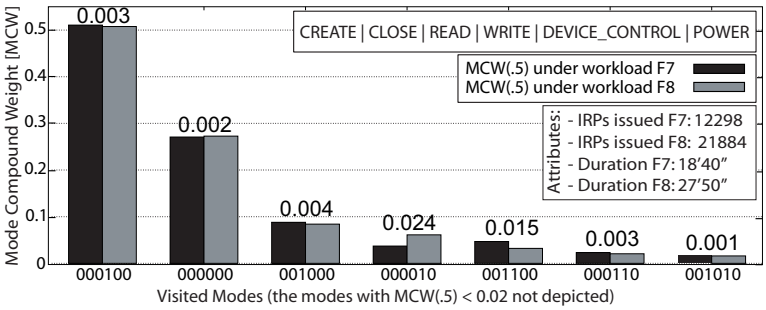


Fig. 28 MCW comparison for F7 and F8

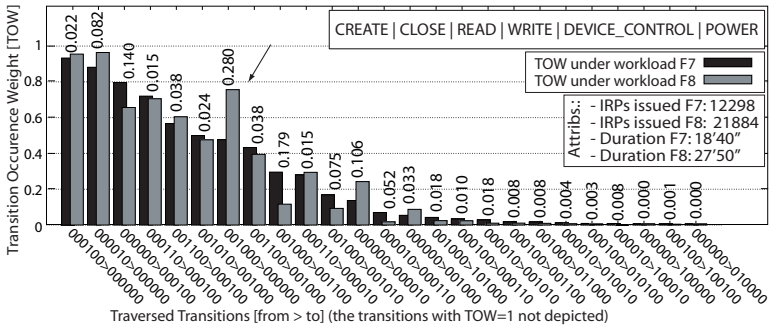


Fig. 29 TOW comparison for F7 and F8

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. Fig. 29 indicates the transition `001000>000000` (the bars pointed by the arrow in Fig. 29) 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 and the DD returns to the *idle* mode. 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.

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 `floppy_XP` 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 objects. 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 Fig. 30 we have used the Euclidean distance among the MCW values of the corresponding modes visited by each workload. Similarly, Fig. 31 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 Fig. 30 indicates that the associated workloads have visited the same modes, generating similar MCW values for each of them.

Fig. 30 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 2), 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 Fig. 30 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 Fig. 28 is visually confirmed by the MDS plot, as the points associated with the two workloads are very close to each other.

To examine how the workloads compare for the traversed edges, in Fig. 31 we have considered as object attributes the TOW values of every transition of the OPs. Here, F7 and F8 are located 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 traverse only a small subset.

As mentioned before, for the MDS plots in figures 30 and 31 we assigned equal weights to modes and transitions, respectively. 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. For instance, Fig. 32 shows the relative similarity among the workloads when only the initial mode (`000000`) and the mode associated with DEVICE_CONTROL (`000010`) are considered as object attributes, all the rest of the modes being ignored (all considered workloads visit these two

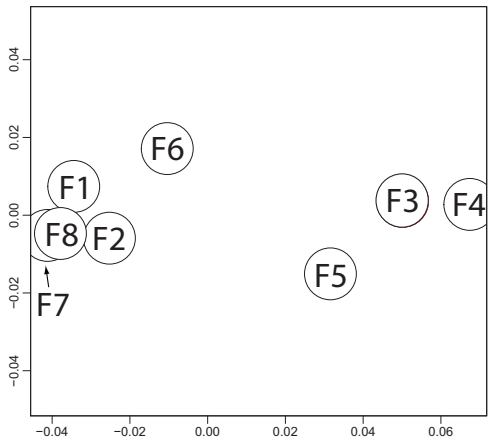


Fig. 30 The distance among workloads (modes only)

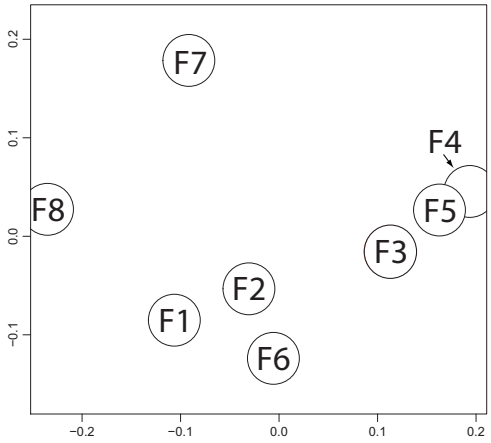


Fig. 31 The distance among workloads (edges only)

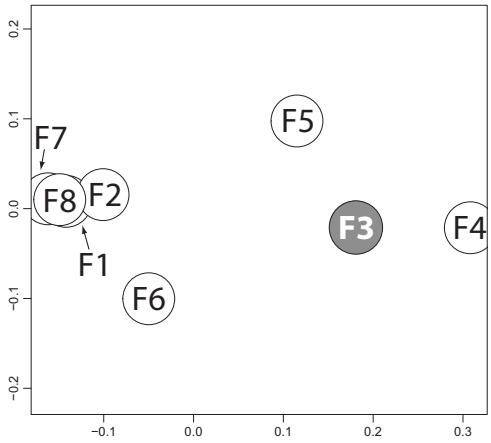


Fig. 32 MDS plot considering two modes only

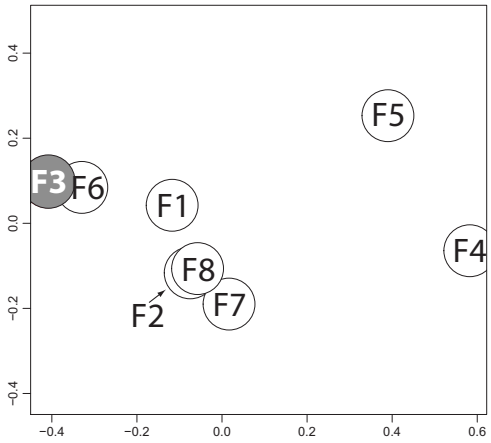


Fig. 33 MDS plot considering two edges only

modes). The MDS plot in Fig. 33 is computed when weight is assigned only to the transitions between the same two modes.

While in Fig. 32 the spatial configuration remains the same as in Fig. 30 (despite the fact that F1, F2, F6 and F7 cluster closer to each other), the positions change dramatically in Fig. 33 as compared to Fig. 31. The closeness between F3 and F6 reveals that DC2 generates an execution pattern which is very similar to the one recorded for F6. 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.

Interestingly, our MDS plots revealed the tendency of DC2 to cluster closer to the workloads that are associated with driver maintenance activities, F4 and F5. Given that DC2 is a tool for testing the robustness and security of drivers, this tendency indicate that the Enable and Disable are exercising driver functionalities considered to be potentially harmful by Microsoft's developers. Moreover, it is reasonable that DC2 shows dissimilarity with the other workloads, which are mostly associated with floppy disk driver's customary operations.

6.4 Test Space Reduction – Tunability for Mode Coverage

To evaluate the test-space reduction enabled by our current profiling approach, we compared it with our previous method (Sârbu et al 2006), considering both modes and transitions that need be covered by a testing campaign. Table 3 lists the overall improvement introduced by the current approach.

First-pass reduction: In figures 34 and 35, for each DD, we have selected the workload that exercised the largest set of modes and transitions, respectively. For instance, for the `floppy_XP` DD we selected the workload F7, as it issues IRPs belonging to five distinct types out of the six types supported by the respective DD. This indicates a *theoretical state space* size of $2^6 = 64$ modes and $6 \cdot 2^6 = 384$ transitions. From this set, only 15 modes (23.44%) and 34 transitions (8.85%) were visited (see Table 2, columns 5–7). In figures 34 and 35, we call this early reduction step *first-pass reduction*.

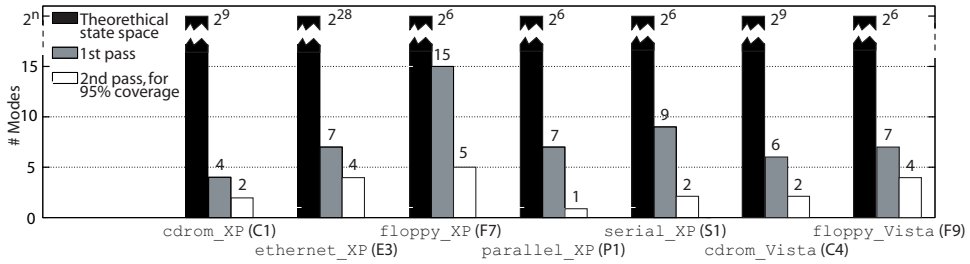


Fig. 34 Test space reduction (modes)

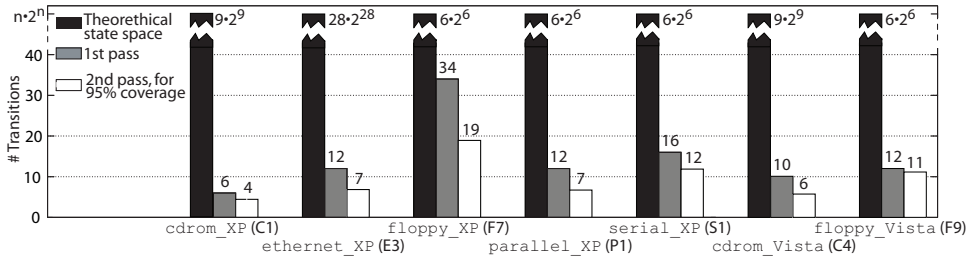


Fig. 35 Test space reduction (transitions)

While the first-pass reduction solely divides the theoretical state space into two classes (visited and non-visited),we additionally utilize 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 34 and 35 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 6.2 and depicted by Fig. 26.

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*" refers to 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 by their MCW weights. Fig. 36 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 test space reduction of 66.66% compared to the first-pass presented in (Sârbu et al 2006). When 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.

Fig. 36 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 34 and 35 depict the second-pass reduction of modes (and transitions, respectively) for 95% testing coverage. Table 3 lists the reduction for each of the workloads considered in figures 34 and 35.

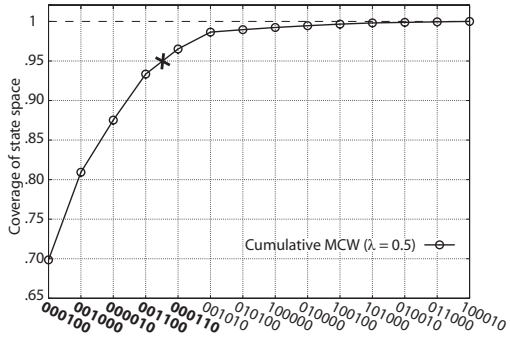


Fig. 36 The cumulative coverage of the driver's modes for F7; modes are ordered in decreasing MCW order from left to right

Table 3 Test space reductions of the second-pass relative to the first-pass ($T = 95\%$)

Quantifier	C1 [%]	E3 [%]	F7 [%]	P1 [%]	S1 [%]	C4 [%]	F9 [%]
MCW	50.00	42.85	66.66	85.71	77.77	66.66	42.86
TOW	33.33	41.66	44.11	41.66	25.00	40.00	8.33

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.

We are currently investigating technical methods to automate test case generation based on the state space identified by our profiling methods. The goal is to take a "real-world" workload and using its OPs to automatically create a set of focused test cases for the considered DD.

7 Experimental Aspects

7.1 Validity Caveats

External threats to validity. We discuss here the issues that can potentially limit the generalization of the experimental approach for a wider population of objects. The drivers chosen for our experiments are WDM-compliant DDs and therefore do not represent a random selection, as they were chosen from the set of drivers installed on the host machine. However, we believe that they are representative for the population of Windows XP DDs as the WDM drivers constitute the main type of drivers available for this OS (Oney 2003). To reduce this threat we selected as diverse DDs as possible (see Table 2). Also, the filter driver used to monitor the selected DD's runtime activity was kept general enough to be ported on virtually any WDM-compliant DD. The representativeness of the workloads chosen to exercise the DDs does not impact the validity of the experiments presented in this paper as we are not attempting to completely profile the field behavior of the DDs. We only propose a general methodology to quantify its behavior. Instead, the workloads listed in Table 2 were selected as they trigger a wide set of specified driver functionalities. However, for obtaining accurate OPs we recommend collecting the OP in the field (e.g., during a beta-testing campaign involving many users); when this is not possible, a wide selection of workloads for the target driver has to be used to ensure accuracy of the OP quantifiers.

The *OP-Builder*, the software tool created for offline log parsing and analysis was found not to scale to extremely large log files. This limitation is due to an inefficient object management of the JVM which leads to memory leaks. However, this has no impact on the results presented in this paper but we are aware that the reusage of our tools in larger DD profiling projects may require handling this issue, possible by running it in other (better) JVMs or, ultimately, porting it to other programming language.

Internal threats to validity. The internal threats to validity of the presented experimental results represent conditions influencing the dependent variables of the experiments. The main threats to internal validity are represented in our experiments by *confounding* and *instrumentation*.

Confounding occurs whenever an extraneous variable changes along with the independent variable, preventing the inference of a causal relationship between the independent and dependent variables. To avoid this threat to the validity of our experiments, we have carefully ensured that no other active process accessed the target DD in the experiment's time window. To detect and filter out the I/O calls not belonging to the considered workload–DD pair before proceeding with OP analysis, we accordingly marked the messages intended for logging.

Instrumentation threat is produced by changes introduced to the studied system by the measurement instrument itself. To reduce this internal threat, we have restarted each DD (the DD's code was removed and then reinstalled in the OS kernel) after each experimental run. Also, to keep our approach as non-intrusive as possible, only the filter driver was inserted into the target OS kernel on the I/O requests' path. With its code written with efficiency in mind, the filter driver only captures, logs and then forwards the IRP traffic. Table 4 lists the average overheads introduced by the filter driver for the drivers presented in this study. Each driver was exercised with the DC2 testing tool from Microsoft. It was run 64 times with and 64 times without the filter driver installed in the driver stack. We measured two temporal parameters of each run: (a) *elapsed time* and (b) *CPU time*. The elapsed time represents the total duration of a run (i.e., calculated from the moment when the process starts until it finishes), while the CPU time is the total time spent by the DC2 process in the CPU (i.e., executing computing-intensive operations).

The results presented in Table 4 show that our filter driver induces a minimal overhead in terms of elapsed time averaging 2.7% increase, despite the fact that the CPU time increase is on average almost 22% higher than without our filter driver installed. However, this apparently high impact is actually insignificant as our workload is mostly I/O intensive, and the CPU time represents only 6% of the total elapsed time. The data presented in Table 4 also show that the increase is similar across all the studied drivers, indicating the fact that the overhead introduced by our filter driver is stable irrespective of the type of the driver that it is installed upon. Interestingly, for the DDs profiled under Vista (`cdrom_Vista` and `floppy_Vista`) the influence of the filter driver was higher than for the corresponding XP DDs.

Table 4 The overheads introduced by the filter driver in terms of elapsed time, process time and system call invocations for all the studied drivers. The data represents averages over 128 runs of each driver, 64 with and 64 without the filter driver installed.

Driver	**Issued IRPs**	**Elapsed Time**			**CPU Time**		
		w/o Filter t_0 **[ms]**	**w/ Filter** t_1 **[ms]**	**Increase** $\frac{t_1-t_0}{t_0} * 100$ **[%]**	**w/o Filter** t_2 **[ms]**	**w/ Filter** t_3 **[ms]**	**Increase** $\frac{t_3-t_2}{t_2} * 100$ **[%]**
`cdrom_XP`	16578	5418	5428	**+0.18**	276	340	**+23.18**
`cdrom_Vista`	4094	1600	1685	**+5.31**	120	140	**+16.67**
`floppy_XP`	10334	3010	3087	**+2.55**	165	208	**+26.06**
`floppy_Vista`	10362	3070	3308	**+7.75**	234	289	**+23.50**
`parallel_XP`	12530	4896	4929	**+0.67**	205	253	**+23.41**
`serial_XP`	2112	1230	1238	**+0.65**	86	98	**+13.95**
`ethernet_XP`	12504	3516	3585	**+1.96**	186	236	**+26.88**
AVERAGE	9787.7	3248.5	3322.6	**+2.72**	181.7	223.4	**+21.95**

7.2 The Effort Required to Profile and Analyze the Operational Activity of Device Drivers

Development Effort. The primary effort in obtaining the experimental data (presented in Section 6) was spent in implementing the filter driver and the *OP-Builder* tool (see Fig. 4). Due to the constraints imposed by WDM and DDK (Oney 2003), the filter driver was programmed using plain C. In contrast, *OP-Builder* was programmed using Java.

The SLOC (source lines of code) metric gives an insight about the effort required to build the tools. SLOC counts the number of physical lines of code, whereas the blank and comment lines are not counted (for a detailed definition see Wheeler (2001)). Hence, the filter driver has a SLOC count of 378, while the *OP-Builder* has a SLOC count of 1317. The required amount of work estimates to approximatively one person–month for the filter driver and a little over four person–months for developing the *OP-Builder* tool (we have used the COCOMO metric (Boehm 1981) to estimate the development cost).

Usage Effort. As the tools developed for monitoring and analyzing purposes are supposed to be widely portable across the population of WDM-compliant DDs as possible, we expect that their re-usage effort is kept to a minimum. No re-compilation of the tools is required for analyzing each DD, the configuration parameters are imple-

mented as editable text files. Hence, the usage effort is represented only by tool configuration. In the case of the filter driver, the name of the target-DD has to be specified in a configuration file, while for the *OP-Builder* tool one must list the supported IRP calls in a text file. Several students were asked to run profiling sessions for several drivers, after they were instructed how to use the tools. Thanks to the high degree of automation and simplicity of the configuration mechanisms, all the students were able to successfully produce OPs for the given set of DDs in a few minutes of work.

Analysis Effort. The *OP-Builder* tool takes log files containing I/O traffic information as inputs and produces OP graphs as outputs. The OP graphs are represented in DOT language, thus they can be viewed using Graphviz (Simionato 2004) or other open-source graph visualization tools, helping further test decisions. The analysis process is extremely fast, the *OP-Builder* tool can process log files at an average speed of 25000 IRPs per second. The longest time required by the log-analysis stage in our experiments was around three seconds, namely for the C1 workload (see Table 2).

7.3 Experimental Issues

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. This happens due to the DebugView kernel debugger (see Fig. 4), 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 dropped. We are currently working on an improved version of the logging mechanism which circumvents this problem. Our 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. Note that the data presented in Table 2 is complete with respect to the tracked IRPs.

Summarizing the experiences gained from our efforts, Table 5 represents a collection of suggestions intended to further improve the testing of DDs by applying the OP introduced quantifiers.

Table 5 Suggestions for proposed metric usage toward improved DD test campaigns.

► **Filter driver:** build a filter driver (or reuse ours) to monitor the I/O traffic between I/O Manager and the DD of interest; keep it as simple as possible, to avoid negative impact on system performance and reliability.
► **Quantifiers:** verify the log-files generated by the filter driver for lost IRPs to ensure the validity of the obtained data; using the validated logs, compute the OP quantifiers (as presented in Section 4) for each mode and transition of the OP.
► **Choosing λ:** λ is a coefficient designed to enable tuning the test priority toward often visited modes or toward modes where the DD spent large time amounts; vary λ according to test interest and use the obtained MCW values to prioritize the test campaign.
► **Seldomly sojourned modes:** the modes with small values of MOW or MTW should not be overlooked in testing, as their activities are usually crucial for DD's service provision, being associated with management of the DD.
► **Find patterns:** carefully inspect the execution pattern to identify I/O sequences relevant for DD testing (and build test cases accordingly).
► **Compare workloads:** to express the relative similarity among several workloads, measure the Euclidean distance of the mode and transition weights and define a equivalence threshold; for weights smaller than the threshold, the workloads are said to be equivalent.
► **Capture field workloads:** when possible, capture the DD's OP in the field and compare it with the workload used in-house for testing to improve the adequacy of the test process.

8 Summary, Discussion and Future Work

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 obtaining 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 currently 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 methodology to gain insight into the behavior of the driver-under-test at runtime. It is not intended as a tool for finding bugs. Its main purpose is to quantify a DD's state sojourn behavior in order to guide testing towards certain subroutines, but it does not 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 "expensive" faults (i.e., the faults with higher probabilities to be activated after the release of the DD).

To substantiate the value and the portability of the presented profiling methodology, various DDs belonging to the latest Windows OS-family (XP SP2 and Vista SP1) were considered for case studies in conjunction with a comprehensive set of performance and stability benchmarks. Moreover, we compared the versions of the same drivers installed under the two OSs by using the same benchmarks and with same configuration parameters. Thus, we observed that the profiled Vista drivers were more complex than their XP counterparts in terms of activated functionality. We believe this added complexity is an artifact of the evolution of XP's WDM specifications into the newer KMDF (Kernel-Mode Driver Framework (Orwick and Smith 2007)). Given the empirical nature of the presented experiments, we are aware that the validity of the results is currently limited to WDM- and KMDF-compliant DDs. Therefore, we are 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 DD 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.

We intend to perform a detailed analysis of the presented state-based driver model in order to answer the question if increasing the detail richness of the collected data would help improve the accuracy of the model while keeping the abstraction level of a black-box. Another current research direction is a quantitative study of driver-relevant workloads that considers using the metrics introduced in this paper to characterize workloads from a DD's perspective. This information would help guiding the choice of an adequate workload for a given testing campaign.

Acknowledgements This research has been supported, in part, by Microsoft Research, NoE ReSIST (European Network of Excellence for Resilience for Survivability in IST) and Deutsche Forschungsgemeinschaft's project TUD GK-MM.

References

Albinet A, Arlat J, Fabre JC (2004) Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In: International Conference on Dependable Systems and Networks (DSN), 2004, pp 867–876

Arlat J, Fabre JC, Rodriguez M (2002) Dependability of COTS microkernel-based systems. IEEE Transactions on Computers volume 51, issue 2:138 – 163, 2002

Avritzer A, Larson B (1993) Load testing software using deterministic state testing. In: International Symposium on Software Testing and Analysis (ISSTA), 1993, pp 82–88, DOI 10.1145/154183.154244

Ball T, Cook B, Levin V, Rajamani S (2004) SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In: Integrated Formal Methods, Springer Verlag, vol. 2999, 2004, pp 1–20

Ball T, Bounimova E, Cook B, Levin V, Lichtenberg J, McGarvey C, Ondrusek B, Rajamani SK, Ustuner A (2006) Thorough static analysis of device drivers. In:Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), ACM, NY, USA, 2006, pp 73–85, DOI http://doi.acm.org/10.1145/1217935.1217943

Boehm BW (1981) Software engineering economics. Prentice-Hall, Englewood Cliffs, N.J., USA, 1981
Chou A, Yang J, Chelf B, Hallem S, Engler DR (2001) An empirical study of operating system errors. In: ACM Symposium on Operating Systems Principles (SOSP), 2001, pp 73–88
Duraes J, Madeira H (2003) Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. IEICE Transactions on Information and Systems 86(12):2563–2570, 2003
Ganapathi A, Ganapathi V, Patterson D (2006) Windows XP kernel crash analysis. In: Large Installation System Administration Conference (LISA), 2006, pp 12–22
Johansson A, Suri N (2005) Error propagation profiling of operating systems. In: International Conference on Dependable Systems and Networks (DSN), 2005, pp 86–95, DOI 10.1109/DSN.2005.45
Leon D, Podgurski A (2003) A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In: 14th International Symposium on Software Reliability Engineering (ISSRE), 2003, pp 442–453, DOI 10.1109/ISSRE.2003.1251065
McMaster S, Memon AM (2005) Call stack coverage for test suite reduction. In: 21st IEEE International Conference on Software Maintenance (ICSM), 2005, pp 539–548, DOI 10.1109/ICSM.2005.29
Mendonca M, Neves N (2007) Robustness testing of the Windows DDK. In: International Conference on Dependable Systems and Networks (DSN), 2007, pp 554–564, DOI 10.1109/DSN.2007.85
Microsoft Corp. (2008) Driver Verifier. URL http://www.microsoft.com/whdc/DevTools/tools/DrvVerifier.mspx
Microsoft (2006) Windows roadmap for drivers, 2006. URL http://www.microsoft.com/whdc/driver/foundation/DrvRoadmap.mspx
Möller KH, Paulish D (1993) An empirical investigation of software fault distribution. In: First International Software Metrics Symposium (METRICS), 1993, pp 82–90, DOI 10.1109/METRIC.1993.263798
Musa JD (2004) Software Reliability Engineering: More Reliable Software Faster and Cheaper, 2nd edn. AuthorHouse, 2004
Musa JD, Okumoto K (1984) A logarithmic Poisson execution time model for software reliability measurement. In: 7th International Conference on Software Engineering (ICSE), 1984, pp 230–238
Nagappan N, Williams L, Osborne J, Vouk M, Abrahamsson P (2005) Providing test quality feedback using static source code and automatic test suite metrics. In: 16th IEEE International Symposium on Software Reliability Engineering (ISSRE), 2005, pp 83–94
Oney W (2003) Programming the MS Windows Driver Model, 2003. Microsoft Press, Redmond, Washington
Orgovan V (2008) Windows Feedback and Reliability - talk at the 19th IEEE International Symposium on Software Reliability Engineering (ISSRE), 2008 URL http://www.csc2.ncsu.edu/conferences/issre/2008/VinceO_ISSRE_2008.pdf
Orwick P, Smith G (2007) Developing Drivers with the Windows Driver Foundation, 2007. Microsoft Press, Redmond, Washington
Rational (2008) Purify. URL http://www-01.ibm.com/software/awdtools/purify
Rothermel G, Untch R, Chu C, Harrold M (2001) Prioritizing test cases for regression testing. Transactions on Software Engineering 26:929–948, DOI 10.1109/32.962562
Russinovich M (2008) Debugview. URL http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx
Simionato M (2004) An introduction to graphviz and dot. URL http://www.linuxdevcenter.com/pub/a/linux/2004/05/06/graphviz_dot.html
Simpson D (2003) Windows XP embedded with Service Pack 1 reliability. Tech. rep., Microsoft Corporation, 2003, URL http://msdn2.microsoft.com/en-us/library/ms838661.aspx
Sârbu C, Johansson A, Fraikin F, Suri N (2006) Improving robustness testing of COTS OS extensions. In: 3rd International Service Availability Symposium (ISAS), Springer Verlag LNCS vol 4328, 2006, pp 120–139, DOI 10.1007/11955498
Sârbu C, Johansson A, Suri N (2008) Execution path profiling for OS device drivers: Viability and methodology. In: 5rd International Service Availability Symposium (ISAS), Springer Verlag LNCS vol 5017, 2008, pp 90–107
Swift MM, Martin S, Levy HM, Eggers SJ (2002) Nooks: an architecture for reliable device drivers. In: Proceedings of the 10th workshop on ACM SIGOPS European Workshop, ACM, NY, USA, 2002, pp 102–107 DOI http://doi.acm.org/10.1145/1133373.1133393
Swift MM, Bershad BN, Levy HM (2005) Improving the reliability of commodity operating systems. ACM Transactions on Computer Systems 23(1):77–110, 2005, DOI 10.1145/1047915.1047919
Zhou F, Condit J, Anderson Z, Bagrak I, Ennals R, Harren M, Necula G, Brewer E (2006) SafeDrive: safe and recoverable extensions using language-based techniques. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), USENIX, Berkeley, CA, USA, 2006, pp 45–60
Weyuker E (1998) Testing component-based software: A cautionary tale. IEEE Software, Issue: 5:54–59, 1998, DOI 10.1109/52.714817
Weyuker EJ (2003) Using operational distributions to judge testing progress. In: ACM Symposium on Applied Computing, ACM Press, New York, NY, USA, 2003, pp 1118–1122, DOI 10.1145/952532.952750
Weyuker EJ, Jeng B (1991) Analyzing partition testing strategies. IEEE Transactions on Software Engineering 17, Issue: 7:703–711, 1991, DOI 10.1109/32.83906
Wheeler DA (2001) More than a gigabuck: Estimating GNU / Linux's size. URL http://www.dwheeler.com/sloc