On-demand Fault Tolerance

Abstract

Support for checkpoint and restore has long been regarded as a useful feature for fault tolerance. However, to date it has not been possible for device drivers because they share state with their devices that is not available through memory. As a result, capturing enough state to restore driver functionality following a failure is difficult. Past approaches instead rely on restart/replay, which can be slow and requires writing wrapper code to log driver/kernel interactions.

In this paper, we present a device state checkpointing mechanism that creates a snapshot of internal device state and can later restore this checkpoint. This mechanism leverages existing power management functionality present in many drivers, and thus requires little implementation effort.

We demonstrate the usefulness of device state checkpoint by building on-demand fault tolerance (ODFT). This mechanism dynamically isolates driver code at the granularity of a single entry point, as compared to the whole driver, and uses checkpoint/restore for recovery if the entry point fails. In evaluation, we show that checkpointing reduces the latency of recovery by 79% compared to restarting the driver, and that ODFT can recover quickly from a wide range of failures with no performance impact when not isolating critical-path code.

1. Introduction

State checkpoint and restore has long been a fundamental mechanism used to provide fault tolerance [8, 41]. However, most checkpointing techniques only capture memory state. As a result, state stored in attached devices, such as configuration settings are not captured. This prevents operating systems from using whole-system memory checkpoints, because device state cannot later be restored. Only in virtualized environments, where virtual devices can explicitly export their state, can a running OS be checkpointed and later restored from memory [45]. Furthermore, the inability to checkpoint device state prevents checkpoint/restore from being applied to device driver fault tolerance.

In most commodity operating systems, third-party driver code executes in privileged mode. Faulty device drivers cause many reliability issues in these drivers [10, 47]. Hence, there has been significant research to tolerate driver failures using programming language and hardware protection techniques [4, 7, 17, 19, 28, 32, 55]. However, much of this work focuses on *detecting* failures and *isolating* drivers from the rest of the system. Few of these systems address how to *restore* driver functionality beyond simply reloading the driver, which may leave applications non-functioning.

The state-of-the-art mechanism for restoring driver functionality, shadow drivers [51], logs state-changing operations at the driver/kernel interface. Following a failure, shadow drivers restart the driver and replay the entire log in order to restore internal driver and device state. This approach poses three problems. First, restarting a driver can be slow (multiple seconds) due to complex initialization code. Second, continuous monitoring of a driver is needed to enable recovery, which imposes a performance cost on the system. Finally, shadow drivers must encode the semantics of the kernel/driver interface, and thus must be written for each class of drivers and updated when the interface changes. In addition, a generic shadow driver cannot correctly restore drivers with proprietary commands. For example, the side effects of `ioctl` commands are often undefined, and thus may have arbitrary effects on driver and device state.

We present a mechanism for capturing the state of a device called *device state checkpointing* and demonstrate its usefulness. This mechanism can be used prior to invoking driver code, to create a checkpoint that can be used to restore the driver after a failure. We leverage existing software triggered power-management code present in most modern drivers to checkpoint and restore device state. With slight modifications, this code can provide all the needed functionality for checkpoint/restore.

Device checkpoint/restore enables a new set of system features to benefit from fast recovery. Whole system checkpoint/restart [36] must restore driver state following a failure. Similarly, migrating virtual machines using pass-through or virtualization-aware devices [25, 38] requires capturing device state at the source and instantiating it at the destination.

We demonstrate the use of device state checkpoints for fault tolerance with an *on-demand fault tolerance mechanism* (ODFT). This mechanism executes selected driver code in isolation in a transaction-like fashion and uses device state checkpointing perform to recovery after a failure. ODFT reduces the cost of driver fault tolerance, as it incurs overhead only when executing code selected for isolation. Overhead can be further reduced by dynamically selecting whether to execute a function in isolation based on its parameters, such as specific `ioctl` commands.

The contributions of this paper is as follows:

- We provide the ability to create device checkpoints on a running system. Along with system memory checkpoints, it has applications in fault tolerance, live migration and fast re-initialization of devices. In a study of six drivers, we show that taking a checkpoint is fast, averaging only 20 μs.
- We build on-demand fault tolerance, a system that dynamically provides fault tolerance to different parts of a driver on a per-request basis. When used with request monitoring/anomaly detection tools, it removes the permanent overhead of fault tolerance that is associated with existing driver reliability solutions. Furthermore, it supports fast and fine-grained failure recovery and we find that using checkpoints to recover reduces the latency of recovery by an average of 79% to 0.16 seconds.
- We evaluate both techniques on six drivers and find that device checkpoint code can re-constructed with very little developer effort without requiring significant re-engineering of drivers and show that 76% of drivers in common driver classes already contain the power management code necessary for generating checkpoint/restore support.

We begin with a discussion of why device checkpoints are important.

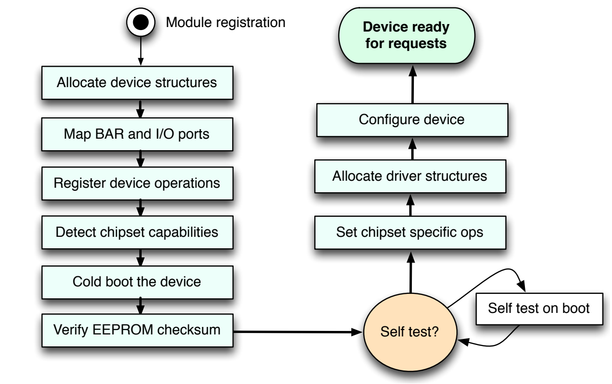


Figure 1. **Modern devices perform many operations during initialization such as setting up kernel and device structures based on chipset and device features, checksumming device ROM data, various device tests followed by driver initialization and configuration.**

2. Motivation

Our work is motivated by limitations in existing mechanisms for capturing and recovering device state and the uses of checkpoints of device state.

Capturing device state. Existing mechanisms for capturing device state have largely been used to recover from failures. Most driver-reliability systems do not try to restore device state and restart failed drivers [21, 52, 56], effectively resetting device state to a known good configuration. Shadow drivers go further by logging driver requests and replaying those requests during recovery [51]. This resets the driver and device to a state functionally equivalent to its pre-failure state.

The restart/replay approach suffers from two issues. First, as shown in Table 7, recovery can be slow and take as long as two seconds to restart some drivers. This latency is high enough that the technique may not be useful in latency-sensitive environments. The primary delay comes from probing the device all over again which cold boots the device and performs the initialization steps as shown in Figure 1. For example, during initialization a network driver probes for the device, verifies EEPROM contents, tests the device, and registers the device with the kernel. In addition, shadow-driver logging imposes a logging cost on every driver operation that increases overhead on high-performance devices such as solid-state drives and high-speed NICs.

Second, shadow drivers require a developer to create a model for each device class indicating which requests to record and how to replay them following a failure. This model must be updated every time the driver/kernel interface changes. Furthermore, shadow drivers cannot restore drivers with behavior outside the specification of a class, such as proprietary `ioctl` commands. Recent work showed that up to 44% of drivers have non-class behavior [26].

Thus, a better approach is needed. In order to re-initialize devices correctly and quickly, we argue that drivers should export checkpoint and restore operations. When used for fault tolerance, checkpoints can reduce latency of recovery by restarting drivers more quickly and reduce the cost and complexity of logging.

Uses for device state checkpointing. In addition to fault tolerance, device state checkpoints have other uses. Table 1 lists five possible uses. Within an operating system, checkpoints support fast reboot after upgrading system software by restoring device state from a checkpoint rather than reinitializing the driver. Similarly, operating systems using non-volatile memory to survive power failures [2, 36] can restart drivers from a checkpoint rather than reinitializing the device.

Fault Tolerance
Device recovery: Current recovery mechanisms require writing wrappers to track all device state and full device restart results in long latency.
OS functionality
Fast reboot: Restarting system requires probing all bus and device drivers.
NVM Operating systems: Providing persistent state of a running system requires ability to checkpoint a running device.
I/O Virtualization
Device consolidation: Re-assignment of passthrough devices across different VMs needs to wait for device initialization.
Live migration: Live migration of pass-through devices converts the millisecond latency of migrations to multiple seconds due to device initialization.
Clone VMs: Ability to launch many cloned VMs very quickly is limited by device initialization.

Table 1. **Uses for fast device state checkpointing.**

In virtualized settings, pass-through and virtualization-aware devices [40] allow drivers in guest operating systems to interact with physical hardware. Device state checkpoints enable virtual-machine checkpoints to a passthrough device [31] and live migration, because the device state from the source can be extracted and restored on identical hardware at the destination. With virtual devices, the latency of live migration can be as low as 60ms [12], so a 2 second delay to initialize a device adds significant downtime.

Finally, device state checkpointing enables dynamic fault tolerance at fine granularity. The overhead of fault tolerance may prohibit its application at all times. If developers or system monitoring tools can identify suspect executions then one can checkpoint device/driver state before problematic code executes and remove the permanent overhead of isolating the complete driver.

Based on the problems with existing driver recovery mechanisms and the uses of device state checkpointing, we propose that drivers should expose an explicit interface to extract device state for checkpointing and reload device state for restore.

3. Device State Checkpointing

To be useful, a device checkpoint mechanism should fulfill the following goals:

1. *Lightweight.* No continuous monitoring or long-latency operations.
2. *Broad.* The mechanism must work with a wide range of devices/drivers, including those with unique behavior.
3. *Consistent.* Drivers are often invoked on multiple threads, and checkpoints must be a consistent view of device state.

While providing such a facility *automatically* for existing drivers is appealing, the wide variety of driver code makes such a goal nearly impossible. Instead, we seek a mechanism that requires a minimum of changes to existing driver and can support the quirks of specific devices while capturing device state.

An appealing approach is to treat devices like memory and copy memory mapped I/O regions. However, reading registers may have side effects such as clear counters. In addition, some devices overlay two logical registers, one for read and one for write, at the same address.

Instead, we take inspiration from code *already present* in many drivers that must perform *nearly the same* task as checkpoint/restore: power management. We next describe how power management for drivers works, and then describe how to reuse the functionality for checkpoint/restore.

3.1 Suspend/Resume Background

Modern operating systems can dynamically reduce their power consumption to provide a hot *standby* mode, also called *suspend to RAM*, which disables processors and devices but leaves state in memory. One major component of reducing power is to disable devices: a spinning hard drive may consume 5 watts when idle [48], while low-power systems can draw below 1 watt [1]. Thus, operating systems direct devices to switch to a low-power state when the system goes into standby mode. The behavior of devices is specified by the ACPI specification for the platform and by buses, such as PCI and USB.

In order to transition quickly between standby and full-power mode, drivers implement a power-management interface to save device state before entering standby mode, and to restore device state when leaving standby mode [14, 15]. These operations must be quick to allow fast transitions. The system-wide suspend-to-RAM mechanism saves the memory state of the driver, and the driver is responsible for saving and restore any volatile device state. While suspended, devices are placed in the D3$_{hot}$ state where they still draw some power. However, on resume drivers move a device to the D0$_{initialized}$ (sometimes via D0$_{uninitialized}$) state, and restore their state through resume code [22].

Drivers implement a power management interface with methods to save and restore state. For example, Linux PCI devices implement these two methods:

```
int (*suspend) (struct pci_dev *dev,
     pm_message_t state);
int (*resume) (struct pci_dev *dev);
```

When saving device state to memory, the driver may invoke the bus to save bus configuration data, as well as explicitly save the contents of select device registers that are not captured by the configuration state. The driver then instructs the device to suspend itself. Simple devices that have no state may simply disable the device.

Upon resume, drivers wake the device and restore their saved state. Since the latency of a system to respond post-resume is critical, the initialization is lightweight compared to restarting the driver, as it assumes the device has not changed. Similar to suspend, simple devices may just re-enable the device without restoring state.

For a system to support standby mode, all drivers must support power management. While not all drivers do (Linux is notorious for incomplete support [33]), it is widely implemented by Windows and MacOS drivers, and support in the Linux drivers is improving.

The functionality provided by driver power management is very similar to what is needed for device state checkpointing. First, it provides the ability to save device state to memory in a way that allows applications to continue functioning. Second, even though the device may continue to receive power, the soft reset that occurs when re-enabling a device ensures that any previous state is replaced by the restored state. Finally, power management is implemented by most commonly used drivers. However, it is not directly usable for checkpointing: power management routines lack the ability to continue executing after suspending a device because the device has been disabled.

3.2 Checkpoint

Device state checkpointing is constructed from a subset of the device suspend support already present in drivers. A device may have many distinct forms of state, each of which require a different mechanism for checkpoint:

1. Device configuration information published through the bus configuration space.
2. Device registers with configuration data specific to the device.
3. Counters and statistics exported by the device and aggregated by the driver.
4. Addresses of memory buffers shared with the driver, such as the DMA ring buffers use by network drivers to send or receive packets.

We note that a checkpoint may not actually contain the full state of the device. Rather, it must contain *enough* information that functionality can later be restored without affecting applications. Thus, device state that can be recreated or recomputed need not be saved. Furthermore, the checkpoint only contains the device state. To be restored properly, it requires a consistent copy of the driver state taken at the same time. Thus, it must be paired with mechanisms such as transactional memory or copy-on-write to save the driver's state.

The configuration state is the easiest to save. Most buses provide a method to save configuration information. For example, PCI drivers in Linux use `pci_save_state`, save includes a set of standard registers and the base address registers (BARs). Each driver, though, must handle the remaining state, separately.

The driver explicitly saves register contents and counters in an internal driver structure. The difference between registers and counters arises during recovery, described below, because counter values cannot be written back to the device.

Memory buffers shared with the device can be recreated. As a result, most device drivers do not include the address of these buffers in a checkpoint. Instead, they free buffers during suspend and re-allocates them during resume.

Figure 2(a) diagrams the tasks performed by suspend and resume, and shows how that code is shuffled to create checkpoint and restore functionality. Of the suspend code, checkpointing reuses all the functionality except detaching the device with the kernel and suspending the device. As an example, Figure 2(b) shows the code to checkpoint the 8139too driver.

It may be necessary to checkpoint a driver while it is in use. Existing suspend routines assume the device is quiescent when the device state is saved. Checkpoint, though, may be called at any time. Thus, it must be synchronized with other threads using the driver. However, because device state checkpointing must be coordinated with other mechanisms for capturing driver state, we do not put synchronization code the checkpoint routine. Instead, we require that the caller of checkpoint synchronize with other threads. In Section 4 we show how this can be done with existing driver locks.

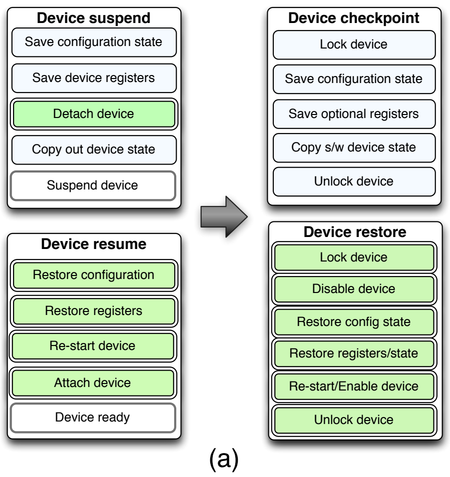
3.3 Restore

The restore operation can be constructed from a mix of suspend and resume code. Normally the resume function is invoked when the device just returned to full power needs to be re-configured. In the case of a checkpoint, though, the device is already running at full power. Thus, resume invokes the bottom half of the suspend routine to disable the device before restoring state.

The restore operation proceeds in four steps:

1. Disable the device to put it in a quiescent, known state.
2. Restore bus configuration state
3. Re-enable the device
4. Restore device-specific state

Figure 2(c) shows the code to restore state for a simple network driver.



```
int rtl8139_checkpoint ( ... )

spin_lock_irqsave
      (&tp->lock, flags);

/* save PCI config state */
pci_save_state (pdev);

/* Copy out device state. */
dev->stats.rx_missed_errors +=
    RTL_R32 (RxMissed);


spin_unlock_irqrestore
       (&tp->lock, flags);
```

(b)

```
int rtl8139_restore (...)

spin_lock_irqsave
         (&tp->lock, flags);

/* disable device */
netif_device_detach (dev);

/* restore bus state */
pci_restore_state (pdev);

/* restart device */
if (!netif_running (dev))
    return 0;
rtl8139_init_ring (dev);
rtl8139_hw_start (dev);

RTL_W32 (RxMissed, 0);

/* re-enable device */
netif_device_attach (dev);
spin_unlock_irqrestore
         (&tp->lock, flags);
```

(c)

Figure 2. Our device state checkpointing mechanism refactors code from existing suspend and resume routine to create checkpoint and restore for drivers. The checkpoint routine only stores a consistent device snapshot to memory while the restore loads the saved state and re-initializes the device. Figures (b) and (c) show checkpoint and restore routines in rtl8139 driver.

Of the four categories of driver state, only configuration state and saved device registers can be reloaded. Counters, which cannot be written back to the device, are restored by adjusting the driver's version of the counter. Typically, the driver will read the device counter and update a copy in memory, and reset the device's counter. To restore the device's counter state, the driver only resets the device's counter; the in-memory copy of the counter must be saved as part of the driver's memory state.

To restore shared buffers, the driver releases existing shared buffers after disabling the device. As part of re-enabling the device, it recreates shared buffers and notifies the device of their new addresses. While this slows restore, it makes checkpoint very efficient because only irretrievable state is saved.

Unlike suspend-resume, it may be useful to use device state checkpointing from interrupt contexts, where sleeping is not allowed. As a result, checkpoint and restore code must convert sleeps to busy waits (`udelay` in Linux) and use memory allocation flags safe for interrupt context (`GFP_ATOMIC` in Linux)

Compared to full-driver restart, resume improves performance because it does not re-invoke device probe, often the lengthiest part of starting a device normally. Furthermore, drivers for newer buses such as USB, and IEEE394 do not restart the device because the bus handles this functionality. This further reduces restore times. For PCI devices, a further optimization is to avoid changing the power mode of the device. The original suspend/resume code puts the device into the $D3_{hot}$ as part of suspend and returns it to the $D0_{initialized}$ state during resume. However, we observed that many drivers do not require actually powering down the device before performing restore. For these drivers, restore can be sped up by skipping these unnecessary power mode changes.

3.4 Summary

Device state checkpointing provides several benefits compared with a logging approach to capturing driver/device state. First, it can be invoked at any time and has no cost until invoked. Thus, it has no overhead for infrequent uses. Second, it handles state unique to a device, such as configuration options. Correct standby operation demands that devices remain correctly configured across standby, and hence drivers must already save and restore any required state. However, device state checkpointing relies on power management code, which may not be present in all devices. It also requires a programmer to implement checkpoint/restore for every driver. We evaluate these concerns in Section 5.

Furthermore, checkpoints only capture device state and not driver state. To enable proper restore, the driver state must also be captured. This can be accomplished with several existing mechanisms, including copy-on-write data structures [42, 50], transactional memory [27], or explicit memory checkpoints [43]. We demonstrate one approach in Section 4.

There is also a risk in utilizing a mechanism for an unintended purpose: the driver continues running following a checkpoint and may thus further modify the device state. In contrast, devices are normally idle between suspend and resume. Thus, it is possible that the state saved is insufficient to fully restore the device to correct operation. However, the power management specifications, requires the drivers to fully capture device state in software since devices can transition to an even lower power state ($D3_{cold}$) where the device is powered off. In such cases, drivers must be able to be restore its original state, following a full reset. Thus, suspend must store enough information to restore from any state. For uses of checkpoint that move state across machines, such as virtual-machine migration, the ability to restore from fully powered off may be necessary as devices may be attached at different I/O memory addresses.

In the case of drivers with persistent internal state, such as disks and other storage devices, restore will only restore the transient device state and not the persistent state, such as the contents of files. As a result, use of checkpoint must be coordinated with higher-level recovery mechanisms, such as Membrane [50], to keep persistent data consistent.

4. On-demand Fault Tolerance

In order to demonstrate the utility of driver state checkpointing, we build a driver fault-tolerance mechanism using checkpoints for recovery called *on-demand fault tolerance* (ODFT). This mechanism provides the ability to dynamically select *which entry points* of a driver, and on *which invocations* to make fault tolerant. Instead of executing the whole driver in isolation, on-demand fault tolerance isolates the execution of a single entry point and recovers from any failures that occur during its invocation. This can greatly reduce the cost of isolating and tolerating faults, because far less code is affected.

This mechanism is useful when specific driver code is known to have problems, such as just-patched code or code with known but unpatched vulnerabilities. Furthermore, ODFT can work with runtime bug isolation tools [30] or security tools [39] and run these requests in isolation without affecting the execution of regular requests. We call the entry points to be isolated *suspect*. The suspect code can be executed in isolation while the remainder of the driver executes in the kernel at full speed. ODFT can also select whether to isolate an entry point at runtime based on its parameters, such as an `ioctl` command code, or enabled at run time through module parameters. The checkpoint and restore routines cannot be isolated and must be trusted.

We perform isolation at the granularity of entry points for two reasons. First, entry points provide a natural atomicity boundary: either the an invocation succeeds, or it fails and has no side effects. Second, drivers often perform synchronization at the granularity of entry points, acquiring locks at the beginning of one and releasing locks at the end. Thus, ODFT reuses existing synchronization mechanisms to ensure that when suspect code runs, no other threads are active in the driver. This ensures that any changes to device state will not be seen until the entry point completes successfully or it fails and recovery completes.

We implemented ODFT for the Linux 2.6.29 kernel. We next describe how ODFT provides isolation, detects failures, and recovers after a failure.

4.1 Isolation

ODFT relies on compile-time software fault isolation (SFI) [54] to prevent buggy code from corrupting the rest of the driver or the kernel. As a driver entry point operates on data shared with the rest of the driver, the SFI mechanism must allow access to such data but prevent its corruption. ODFT therefore executes isolated code on a *minimal copy* of driver and kernel, which is a copy of data it references but not entire structures. If the entry point does not fail, ODFT merges the copy back into the real driver and kernel structures. On a failure, the copy is discarded. This in effect executes the suspect entry point as a transaction using lazy version management [27]. However, not all data can be copied: structures shared with the device, though, such as network transmit and receive rings, cannot be copied because the device will not share the copied structure Instead, ODFT grants suspect code direct read and/or write access to these structures and relies on device-state checkpointing to restore these structures following a failure.

4.1.1 Software Fault Isolation

As ODFT targets open-source Linux device drivers, we implement SFI using a source-code rewriting tool called *ODFT Isolator* written in CIL [37] that performs static analyses and generates isolation code into the driver.

Isolator generates an additional driver module called the *SFI module* that contains a copy of the suspect entry points and all functions transitively called from those functions, instrumented for SFI. In addition, Isolator generates a new version of the driver that invokes the SFI module entry points. At the top of the existing entry points, Isolator inserts a test to see whether to execute normally or in isolation, and if so invokes the SFI module. The decision to invoke a particular entry point in isolation can also be made statically using attribute `__attribute((isolate))` or can be made by adding detecting buggy code with a static-analysis tool. Isolator manages resource allocation, synchronization and I/O across the two copies.

Figure 3 shows the components of ODFT. In addition to producing the SFI driver code, Isolator also produces communication code that invokes the SFI driver and copies in the minimal driver and kernel state needed by the suspect entry points, copies out any changes made by the SFI driver, and initiates recovery following a detected failure.

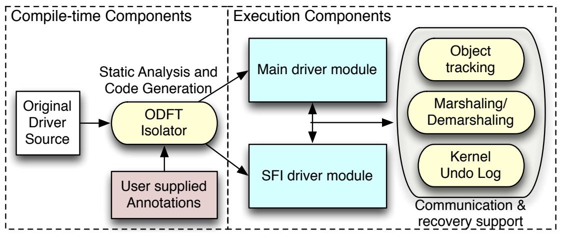


Figure 3. On-demand fault tolerance replicates driver entry points into a normal driver and an *SFI driver module*. A runtime support module provides communication and recovery support.

Isolator uses CIL's memory tracking module [37] to instrument all memory references in the driver. It inserts a call to our `memcheck` function that verifies the target of a load/store is valid. If not, it detects a failure and invokes the recovery mechanism.

The `memcheck` routine consults *range table* to verify memory references. This table contains the addresses and lengths of copied data structures and buffers shared with the device. The range table and object tracker are created on every invocation of a suspect entry point and flushed when on return.

We do not add all local variables to the range table because we trust the compiler to generate correct code for moving variables between registers and the stack. However, if the driver ever takes the address of a local variable, or it creates an array as a local variable then Isolator adds a call in the instrumented SFI driver to add the variable's address and length to the range table and remove it from the range table when the variable goes out of scope. Similarly, we trust the compiler to produce valid control transfers and do not instrument branch or call instructions.

4.1.2 Communication code for entry points

ODFT Isolator generates *stub code* to invoke suspect entry points that copies into and out of the driver. Similar to RPC stubs, these stubs create a copy of the parameters passed to the suspect code, but also copy any driver or kernel global variables it uses. When the suspect entry point completes, stub code copies modified data structures and return values back to the regular driver and kernel.

Isolator automatically identifies the minimal data needed for an entry point through static analysis. This includes the structure fields from parameters referenced by the entry point or functions it calls plus fields of global variables referenced. An alternative approach would be to rely on transactional-memory techniques to dynamically create a copy of data as it is referenced, which may have lower copying costs but higher run-time costs [3]. As they copy data, stubs updates the range table address and length of each object. For objects that cannot be copied (such as those shared with the device), stubs fill in the existing address of the field, its length, and whether the entry point has read, write, or read/write access.

If suspect code callbacks invoke the kernel, Isolator generates stubs for kernel functions that copy parameters to the kernel and copies kernel return values back to suspect code. The SFI driver may pass in fields from its parameters to the kernel as arguments. To avoid creating a new copy of these fields, as would be done by RPC, ODFT maintains an *object tracker* that maps the address of kernel and regular driver objects to the address of objects in the SFI driver. Stub code consults the object tracker when calling into the kernel to determine whether arguments refer to a new or existing object. If an object exists, stubs, copy the argument back to the existing object and otherwise temporarily allocates a new object. To support recovery, stubs may generate an entry in the *kernel undo*

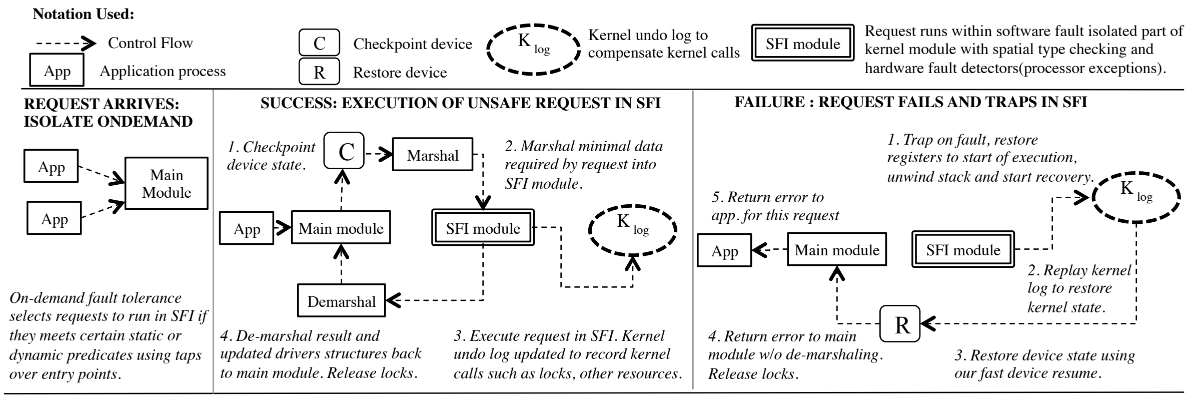

Notation Used:
Control Flow
App Application process
C Checkpoint device
R Restore device
K log Kernel undo log to compensate kernel calls
SFI module Request runs within software fault isolated part of kernel module with spatial type checking and hardware fault detectors(processor exceptions).
REQUEST ARRIVES: ISOLATE ONDEMAND
App
App
Main Module
On-demand fault tolerance selects requests to run in SFI if they meets certain static or dynamic predicates using taps over entry points.
SUCCESS: EXECUTION OF UNSAFE REQUEST IN SFI
1. Checkpoint device state.
C
Marshal
2. Marshal minimal data required by request into SFI module.
App
Main module
SFI module
K log
Demarshal
4. De-marshal result and updated drivers structures back to main module. Release locks.
3. Execute request in SFI. Kernel undo log updated to record kernel calls such as locks, other resources.
FAILURE : REQUEST FAILS AND TRAPS IN SFI
1. Trap on fault, restore registers to start of execution, unwind stack and start recovery.
K log
5. Return error to app. for this request
App
Main module
SFI module
2. Replay kernel log to restore kernel state.
R
4. Return error to main module w/o de-marshaling. Release locks.
3. Restore device state using our fast device resume.

Figure 4. **Execution of requests in SFI module during successful and failed scenarios.**

log, described below, that reveres the side effects the kernel calls made by a failed entry point.

The stub code must know the layout of data structure and whether data is shared with the driver in order to correctly copy data. As driver code often contains ambiguous data structures such as `void *` pointers contained list pointers (*e.g.*, `struct list_head`), we rely on programmer-provided annotations to disambiguate such code [56]. These annotations also declare which structure fields or parameters are shared with the device and should not be copied. In Section 5, we evaluate the difficultly of providing annotations.

Some driver functions trigger synchronous callbacks. For example, the `pci_register_driver` function causes a callback on the same thread to the driver's `probe` function. ODFT treats the callback as a *nested transaction*: it causes another isolated call operating on a second copy of the data.

4.1.3 Resource Access from SFI module

Some resources cannot be copied into the driver because they attach additional semantics or behavior to memory.

I/O memory. Driver entry points may communicate with the device by writing to I/O memory. Stubs grant the SFI driver read/write access to memory-mapped I/O regions and memory shared with the device via `dma_alloc_coherent`. ODFT isolator identifies these regions with annotations and creates stubs that grant drivers direct read/write access. To synchronize device access with other threads in the driver, ODFT re-uses existing device locks. For example, existing device locks in a driver protect against conflicting configuration operations, or operations like resetting the device when I/O operations are in progress. This ensures that we do not corrupt device state assumed by another thread in progress when we reset device state in case of a failure.

Locks. Drivers may synchronize with other threads with spin locks and mutexes. Stubs pass locks through without copying but only allow read access. Suspect code must call back into the kernel to acquire a lock. After acquiring a lock, the SFI driver copies its parameters back from the kernel. The stub code for kernel locking routines add a *compensation entry* to the kernel undo log to release the lock after a failure. To ensure that changes made by suspect code are not seen by the rest of the kernel, the lock stubs defer releasing locks until after the entry point returns to the kernel.

Memory allocation. Stubs for allocators invoke the kernel allocator, add the returned memory region to the range table and generate a compensation entry to free the memory on failure. The newly allocated memory is not copied into the driver because its contents do not need to be preserved.

4.2 Failure Detection

In addition to protecting the kernel and regular driver code from corruption, isolation provides the primary means to detect failures. ODFT's SFI mechanism implements *spatial memory safety* [34]: every memory reference must be within an object made accessible during the copy process. Thus, references outside the range table indicate a failure.

Stubs can detect additional failures when copying data back to the kernel. For example, if the driver writes an invalid address into a data structure, the copying code will dereference that address and generate an exception.

We also modified the Linux kernel exception handlers to detect unexpected traps from the SFI driver as failures. If one occurs, handler sets the instruction pointer to the recovery routine. This is the only change to the Linux kernel, and required only 38 lines of code.

The detection mechanisms may miss several categories of failures. First, if the driver violates its own data structure invariants, stubs may not detect the problem. Recent work on identifying and verifying data structure invariants could detect these faults [6]. For example, if a suspect entry point sets a flag indicating that a field is valid but does not set the field, then corruption will leak out of the SFI driver. Second, the driver does not provide strong type safety, so the driver may assign a structure field to the wrong type of data. While this may be detected when stub copy data, it is not guaranteed. Finally, ODFT does not enforce kernel restrictions on the range of scalar values, such as valid packet lengths.

4.3 Recovery

When a failure is detected, stubs call a recovery routine that is responsible for restoring correct driver operation.

Failure anticipation. To prepare for an eventual recovery, stubs create a device checkpoint before invoking a suspect entry point. They invoke the `checkpoint` routine, as described in Section 3. In addition, stubs for kernel functions log compensation entries to undo their effects in the kernel undo log. Driver state is not explicitly checkpointed; instead suspect code operates on a copy of driver state. In addition, the stub saves its register state, allowing a jump right into the stub if the driver fails.

Recovery steps. The recovery routine restores driver operation through a sequence of steps as shown in Figure 4:

1. *Unwind thread.* If not already in the stub, the instruction pointer is set to the address of the recovery code in the entry point's stub, which reloads the saved registers. As noted above, nested calls to drivers are logically handled as separate transactions, so there is no need to unwind the thread to the outermost entry point.
2. *Restore device state checkpoint.* The stub recovery code calls the driver's `restore` routine to restore the device state.
3. *Free call state.* All temporary structures created for the suspect entry point call, such as the range table, object tracker, and copies of kernel/driver structures are released.
4. *Release locks.* Any locks acquired before or during the call to the SFI driver are release, allowing other threads to execute.

If a driver entry point fails, the stub returns an appropriate error indicator, such as a NULL pointer or an error code, and relies on higher-level code to handle the failure. As only the single entry point fails, but all application state relating to the device, such as open handles, remain valid, this has little impact on applications.

Compared to other driver isolation systems, the recovery process is much simpler because only one thread is affected, so other threads are not unwound. In addition, the driver state is left unmodified, so it is not saved and restored. Finally, device state is restored quickly from a checkpoint rather than by replaying a log. Hence, we see that checkpointing device state results quicker and simpler recovery semantics for driver recovery.

4.4 Implementation effort

ODFT consists minimal modifications to the kernel exception handlers (38 lines of code), a kernel module containing the object tracker, range table, and recovery support, and the Isolator tool in OCaml. The module is 1200 lines of C code, and Isolator is 9700 lines of OCaml that implement: SFI isolation (400 lines), stub generation (7800 lines), and static analysis of references to parameter fields (1500 lines). In comparison, Nooks adds 23,000 lines of *kernel code* to isolate and reload device drivers and shadow drivers add another 1,100 lines of code for recovery.

5. Evaluation

We implemented device state checkpoint and on-demand fault tolerance for the Linux 2.6.29 kernel for six drivers across three buses.

5.1 Device State Checkpointing

We first evaluate device state checkpointing against our goals in Section 3:

1. *Breadth.* How many existing drivers support power management? We identify how broadly we can apply device state checkpointing.
2. *Implementable.* We measure programmer effort in implementing checkpoint and restore.
3. *Lightweight.* How fast is checkpoint/restore compared to a full driver restart?

All experiments were performed on an dual-core 3 GHz Intel Pentium D with 1GB of memory and an Intel 82541PI gigabit NIC. We note this is a somewhat old machine and a more recent machine would likely show lower performance costs.

Support for power management. Device state checkpointing relies on existing power-management code. We measure how broadly it applies to Linux drivers by counting the number of drivers with power-management support. While modern ACPI-compliant systems require that all devices support power management, many legacy drivers do not.

Class	Bus	Drivers reviewed	Drivers with PM
net	PCI	104	68 (65%)
net	USB	32	27 (84%)
net	PCMCIA	4	4 (100%)
sound	PCI	72	63 (88%)
sound	USB	3	1 (33%)
sound	PCMCIA	2	2 (100%)
ATA	PCI	61	45 (74%)
SCSI	USB	1	1 (100%)
SCSI	PCMCIA	5	5 (100%)
Total	**-**	**284**	**216 (76%)**

Table 2. **List of drivers with and without power management as analyzed with static analysis tools. USB devices (audio and storage) support hundreds of devices with a common driver, and provide support for suspend resume.**

Driver	Recovery additions	
	LOC Moved	LOC Added
8139too	26	4
e1000	32	10
r8169	17	5
pegasus	22	4
ens1371	16	6
psmouse	19	6

Table 3. **Developer effort for exporting checkpoint/restore driver callbacks.**

We perform a simple static analysis over all network, sound, and storage drivers using the PCI, USB, and PCMCIA bus in Linux 2.6.37.6. The analysis scans driver entry points and identifies power management callbacks. Table 2 shows the number of drivers scanned by class and bus and the number that support power management.

Overall, we found that 76% of the drivers support power management. Of the drivers that do not support power management, most were either very old, from before Linux supported power management, or worked with very simple devices. Only two modern devices, both Intel 10gb Ethernet cards, did not provide suspend/resume. Thus, we find that nearly all modern devices support power management and can therefore support device state checkpointing.

Implementability. We evaluated the ease of implementing device state checkpointing by adding support to the six drivers listed in Table 3. For each driver we show the amount of code we copied from suspend/resume to create checkpoint/restore as well as the number of new lines added. On average, we moved 22 lines code and added six lines. The new code adds support for checkpoint/restore in interrupt contexts and to avoid nested locks when the routines are invoked with a lock held. Even though device state checkpointing requires adding new code, the effort is mostly moving existing code. In comparison, implementing a shadow driver requires (i) building a model of driver behavior and (ii) writing a wrapper for every function in the driver/kernel interface to log state changes.

Checkpoint/restore latency. The primary goal of device state checkpointing is to improve restore time compared to restarting the driver. Table 4 shows the latency of a checkpoint or restore for the same six drivers. Checkpointing is very fast, taking only 20μs on average and 33μs at worse. Thus, it is fast enough to be called frequently, such as before the invocation of most driver entry points. Restore times are longer, with a range from 30μs for the r8169 network driver to 390ms for psmouse. USB drivers store

Driver	Class	Bus	Checkpoint times	Restore times
8139too	net	PCI	33μs	62μs
e1000	net	PCI	32μs	280ms
r8169	net	PCI	26μs	30μs
pegasus	net	USB	0μs	4ms
ens1371	sound	PCI	33μs	111ms
psmouse	input	serio	0μs	390ms

Table 4. Latency for device state checkpoint/restore.

less state because the USB bus controller stores configuration information instead of the driver. Thus, during a normal resume the bus restores configuration state before calling the driver to resume. They psmouse represents a legacy device and does not support suspend/resume. Instead, we re-use existing device code to reset the mouse, during a system suspend using its private structure information.

Overall, these three results demonstrate that device state checkpoint can be implemented in the majority of modern drivers with little programmer effort and provides high performance. We address the question of whether it actually works in the next section, when we evaluate on-demand fault tolerance.

5.2 On Demand Fault Tolerance

We evaluate the benefit in handing faults and the cost in performance and complexity of ODFT.

1. *On-demand.* Is selectively isolating entry points useful? We evaluate whether suspect entry points can be identified in drivers and whether they reduce the amount of code isolated.
2. *Fault resilience.* What failures can on-demand fault tolerance can handle? We evaluate on-demand fault tolerance using a series of fault injection experiments and report our results.
3. *Recovery time.* What is the downtime caused by a driver failure? We compare the time taken by ODFT to restore the device and cleanup the failed driver thread state with the time taken to unload and reload a driver.
4. *Performance.* What is the performance loss of on-demand fault tolerance on steady-state operation? We report the performance cost for applying ODFT on support and core I/O routines.
5. *Developer effort.* What is the overhead to the developer to enforce isolation in the system? We measure the effort needed to annotate a driver for isolation.

On-demand fault tolerance We evaluate whether selectively isolating specific entry points is useful by looking for evidence that driver bugs are confined to one or a few entry points. In order to have a large data set, we use a published list of hardware dependence bugs which represent one of the larger number of unfixed bugs in the drivers [13]. We were able to map these bugs in 210 drivers (541 total bugs) to our kernel under analysis. For each driver, we calculate the number of entry points and the fraction of code in the driver that must be isolated. If the functions with bugs are reachable through a large number of entry points, then full driver isolation is more useful than per-entry point isolation. For example, if a bug occurs in a low level read routine, then the bug will affect a large number of entry points.

We find that the bugs are reachable through 643 entry points, for an average of 3 per driver. As a comparison, these drivers have a total of 4460 entry points (21 per driver), so only 14% of entry points must be isolated. The code reachable from these entry points comprises only 18% of the code in these drivers. These results indicate that at least some classes of driver bugs are confined to a single entry point, and therefor on-demand fault tolerance can reduce the cost of fault tolerance as compared to isolating the entire driver.

Fault Type	Description of fault
Corrupt pointers	Dynamic: Corrupt all pointers referenced in a function to random values.
Corrupt stack	Dynamic: corrupt execution stack by copying large chunks of data over stack variable addresses.
Corrupt expressions	Static: corrupt arithmetic instructions by adding invalid operations (like divide by zero).
Skip assignment	Static: remove assignment operations in a function.
Skip parameters	Dynamic: zero incoming parameters in a function.

Table 5. Faults injected to test failure resilience represent runtime and programming errors. Dynamic faults are inserted by invoking an `ioctl`, and static faults by making an additional pass to inject faults while converting the driver to support ODFT.

Driver	Injected Faults	Benign Faults	Native crashes	ODFT crashes
8139too	43	0	43	NONE
e1000	47	0	47	NONE
ens1371	36	0	36	NONE
pegasus	34	1	33	NONE
psmouse	22	1	21	NONE
r8169	46	0	46	NONE
Total	**258**	**2**	**256**	**NONE**

Table 6. Fault injection table with number of unique faults injected per driver. ODFT is able to correctly restore the driver state and device state in every case.

Fault Resilience. We evaluate how well on-demand fault tolerance can handle driver bugs using a combination of dynamic and static fault injection over six drivers. These tests evaluate both the ability of on-demand fault tolerance to isolate and recover driver state as well as the ability of device state checkpointing to restore device functionality.

Static fault injection modifies the driver source code to emulate programming bugs, while dynamic fault injection modifies driver data while running to emulate runtime errors. Table 5 describes the types of faults inserted in the SFI module. We perform a sequence of trials that test each fault site separately.

During each experiment, we run applications that use the driver to detect whether a driver failure causes the application to fail. For network, we use as `ssh`, and `netperf`; for sound we use `aplay`, `arecord` from the ALSA suite. We tested the mouse by scrolling the device manually as we performed the fault injection experiments. After each injection experiment, we determine if there is an OS/driver crash or the application malfunctions. We re-invoke the failed entry point without the fault to ensure that it continues to work, and that resources such as locks have been appropriately released.

We injected a total of 258 unique faults in native and ODFT drivers. Table 6 shows the number of faults injected for every driver and the outcome. For the native driver, all but two faults crashed the driver or resulted in kernel panics. The two benign faults were missing assignments.

In contrast, when we injected faults into driver entry points protected by ODFT, the driver and the OS remain available for every single fault. Furthermore, in every case the applications continue to execute correctly following the fault. For example, the sound ap-

Driver	Class	Bus	Restart recovery	ODFT recovery	Speedup
8139too	net	PCI	0.31s	0.07ms	4400
e1000	net	PCI	1.8s	295ms	6
r8169	net	PCI	0.12s	0.04ms	3000
pegasus	net	USB	0.15s	5ms	30
ens1371	sound	PCI	1.03s	115ms	9
psmouse	input	serio	0.68s	410ms	1.65

Table 7. **Recovery times with on-demand fault tolerance when compared with the existing techniques of unloading and reloading the whole driver. Restart based recovery may require additional time to replay logs running over the lifetime of the driver..**

plication `aplay`, skips for few milliseconds during driver recovery but continued to play normally. The millisecond disruption is noted by an "`ALSA buffer underrun`" message in the shell.

We also verify that internal driver and device state is correctly recovered using the `ethtool` interface for network drivers. We find that when failures happen during a call to change configuration settings, re-reading settings after a crash always returns the correct values.

Recovery time. A major benefit of device state checkpointing and ODFT against past recovery techniques is the speed of recovery. Table 7 lists the time taken by driver to recover using ODFT and natively by unloading and reloading the driver. We measure recovery times by recording the time from detection of failure to completion of the restore routine. Overall, ODFT is between 1.6 and 4400 times faster than restart recovery, and between 145ms and 1.5s faster. The drivers with the largest speedup have complicated probe routines that are avoided by restoring from checkpoint. Hence, ODFT provides low-latency recovery and thus is suitable for a wide variety of applications.

Performance. The primary cost of using ODFT is the time spent in copying data in and out of the SFI module and creating device checkpoints. We measure performance with a gigabit Ethernet driver, as it is may send or receive receive more than 75,000 packets per second. Thus, the overhead of ODFT will show up more clearly than on low-bandwidth devices.

We evaluate the runtime costs of using ODFT and regular versions of driver in six configurations:

1. *Native:* Unmodified e1000 driver.
2. *ODFT $_{static}$:* Statically choose 75% of code (all off I/O-path) to isolate.
3. *ODFT $_{dynamic}$:* Dynamically choose at runtime whether to isolate off-I/O-path code.
4. *ODFT $_{1/10}$:* Isolate I/O path code for every 10th packet.
5. *ODFT $_{all}$:* Isolate I/O path code for every packet

The *dynamic* experiment measures the additional cost of choosing at runtime whether to invoke the regular or SFI version of a function. The *1/10* test measures the performance benefit of dynamically choosing whether to isolate based on parameters, such as packets from an untrusted process or destined for an untrusted network. Finally, the *all* test represents the worst case of invoking the SFI module on the I/O path for a high-bandwidth device.

We measure performance with netperf [24] by connecting our test machine to a to another machine with 1.2 GHz Intel Celeron processor and a Belkin NIC with a crossover cable. Table 8 reports the average of 5 runs.

In the *static* and *dynamic* tests where code off the I/O-path code is isolated, performance and CPU utilization are unaffected. These results demonstrate that ODFT achieves the goal of only imposing

Intel Pro/1000 gigabit NIC (E1000)

System	Throughput	CPU Utilization
Native	751.5 Mb/s	5.1%
ODFT $_{static}$	751.1 Mb/s	5.0%
ODFT $_{dynamic}$	751.1 Mb/s	5.2%
ODFT $_{1/10}$	745.3 Mb/s	5.4%
ODFT $_{all}$	454.0 Mb/s	50.0%

Table 8. **TCP streaming send performance with netperf for regular and ODFT drivers with automatic recovery mechanism for the E1000 driver.**

Driver	Driver LoC	Isolation annotations	
		Driver Annotations	Kernel Annotations
8139too	1904	15	20
e1000	13973	32	
r8169	2993	10	
pegasus	1541	26	12
ens1371	2110	23	66
psmouse	2448	11	19

Table 9. **Annotations required in ODFT isolation mechanisms for correct marshaling. Kernel annotations are common to a class driver annotations are specific to a single driver.**

a cost on isolated code. The added cost of dynamically choosing to isolate is in the noise. For the *1/10* test that isolates entry points on the I/O path (the packet send routine) for every tenth packet, bandwidth dropped 0.6% and CPU utilization increase negligibly. Thus, selectively applying isolation, even on critical I/O paths, can have a small impact.

The performance only drops appreciably when we isolate critical path code on every request since we copy shared driver and kernel data across modules for *each* packet being transmitted. In the *all* test, the system consumes 100% of the CPU on one core (50% total CPU utilization), and as a result bandwidth drops by 39%. This occurs because interrupts on other processer must spin until the copying and call completes, which drives up CPU utilization. ODFT is designed to limit isolation costs to specific requests and hence the cost of isolation is high because it needs to setup isolation (create copies) as each packet requests isolation. For such frequent calls where the module needs to be isolated at all the times (as opposed to on-demand), using hardware support (such as IOAT [23]) can reduce isolation overhead. Furthermore, to provide permanent isolation techniques that isolate the entire driver, such as Nooks [52] and SUD [4], may prove better because they copy less data per request. However, these techniques impose a permanent cost over all requests.

Overall, these results that ODFT performs well for the vast majority of devices that are low bandwidth, and can also be applied at low cost to high bandwidth devices off the I/O path. In addition, if the driver can dynamically detect which calls may be error prone to reduce the frequency of calls to the SFI driver, it can also be applied on the I/O path.

Developer effort. The primary development cost in using ODFT is adding annotations, which describe how to copy data between the kernel and the SFI module. Table 9 shows the number of annotations needed to apply ODFT to every function in each of the tested drivers. We separate annotations into *driver annotations*, which are made to the driver code, and *kernel-header annotations*, which are a one-time effort common to all drivers in the class.

Overall, drivers averaged 20 annotations, with more annotations for drivers with more complex data structures. Most driver classes required 20 or fewer kernel-header annotations except for sound drivers, which have a more complex interface and required 66 annotations.

Thus, the effort to annotate a driver is only modest, as annotations touch only a small fraction of driver code. In comparison, SafeDrive [56] changed 260 lines of code in the e1000 driver for isolation and another 270 lines for recovery. Nooks [52] required 23,000 lines of code to isolate and reload drivers. Thus, these small annotations to drivers may be much simpler than adding a large new subsystem to the kernel.

6. Related work

On-demand fault tolerance draws inspiration from past projects on driver reliability, program partitioning and software fault isolation systems.

Device driver recovery. Prior driver-recovery systems, including Nooks [52], Shadow drivers [51], SafeDrive [56] and Minix 3 [21] all unload and restart a failed driver. In contrast, ODFT takes a checkpoint prior to invoking the driver, and then rolls back to the most recent checkpoint, which is much faster. CuriOS provides transparent recovery and further ensures that client session state can be recovered [16]. However, CuriOS is a new operating system and requires specially written code to take advantage of its recovery system, while ODFT works with existing driver code in existing operating systems. ReViveI/O [35], and similar systems [44] provide whole-system checkpoint/restore by buffering I/O and only letting it reach the device after the next memory checkpoint. However, this approach does not work with polling, where I/O operations cannot be buffered and applied later.

Driver isolation systems. Driver isolation systems rely on hardware protection (Nooks [52] and Xen [18]) or strong in-situ detection mechanisms (BGI [7], LXFI [32] and XFI [53]) to detect failures in driver execution. However, in latter systems if the failure is detected *after* any state shared with the kernel has been modified then these systems cannot rollback to a last good state. Other driver isolation systems such as SUD [4] and Linux user-mode drivers [29], require writing class-specific wrappers in the kernel that are hard to maintain as the kernel evolves. ODFT differs from existing isolation systems by providing fine-grained isolation semantics and limits the runtime overheads only to suspect code. Prior work on fast software fault isolation, such as BGI [7], would benefit ODFT by improving its performance on I/O-intensive workloads.

Software fault tolerance. Existing SFI techniques use programmer annotations (SafeDrive [56] or API contracts (LXFI [32]) to provide type safety. XFI [53] transforms code binaries to provide inline software guards and stack protection. In contrast, ODFT operates on source code and allows drivers to operate on a copy of shared data. ODFT marshals minimum required data and in buffers and uses range hash to provide spatial safety.

Transactional kernels. ODFT executes drivers as a transaction by buffering their state changes until they complete. VINO [49] similarly encapsulated extensions with SFI and used a compensation log to undo kernel changes. However, VINO applied to an entire extension and did not address recovering device state. In addition, it terminated faulty extensions, while most users want to continue using devices following a failure. ODFT is complementary to other transactional systems such as TxLinux [46], which provide transactional semantics for system calls. These techniques could be applied to driver calls into the kernel instead of using a kernel undo log of compensation records.

Program Partitioning Program partitioning has been used for security [5, 9] and remote code execution [11]. Existing program partitioning tools statically partition user mode code or move driver code to user mode [20]. ODFT is the first system to partition programs within the kernel and is hence able to provide partitioning benefits to kernel specific components such as interrupt delivery and critical I/O path code. Furthermore, instead of partitioning code in any one domain, ODFT replicates its entry points and decides on a runtime basis whether a particular thread should run in isolation.

7. Conclusions

In this paper, we make a case that devices drivers should support device checkpoint/restore facilities. Checkpoints are useful for numerous applications such as on-demand fault isolation, OS migration, and persistent operating systems. This functionality is often considered to require a significant re-engineering of device drivers. However, we demonstrate that this functionality is already provided by suspend/resume code. Checkpoint/restore enables new applications, such as fine grained fault tolerance using ODFT, which dynamically isolates entry points and quickly restores driver and device functionality in case of failure.

References

[1] Apple Inc. 11-inch macbook air environmental report. `www.apple.com/macbookair/environment.html`, 2011.

[2] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proc. of the Thirteenth IEEE HOTOS*, 2011.

[3] A. Birgisson, U. E. Mohan Dhawan, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, Oct. 2008.

[4] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *USENIX ATC*, 2010.

[5] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. of the 13th USENIX Security Symposium*, 2004.

[6] S. Butt, V. Ganapathy, M. Swift, and C.-C. Chang. Protecting commodity OS kernels from vulnerable device drivers. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, Dec. 2009.

[7] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *SOSP*, 2009. ACM.

[8] K. Chandy and C. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21(6):546–556, June 1972.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. of the 21st ACM SOSP*, 2007.

[10] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proc. of the 18th ACM SOSP*, 2001.

[11] B. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *Proceedings of the 12th conference on Hot topics in operating systems*. USENIX Association, 2009.

[12] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005.

[13] J. Corbet. Trusting the hardware too much. `http://lwn.net/Articles/479653/`. LWN February 2012.

[14] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly Associates, Feb. 2005.

[15] M. Corp. Power management and ACPI - architecture and driver support. `msdn.microsoft.com/en-us/windows/hardware/gg463220`.

[16] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proc. of the 8th USENIX OSDI*, December 2008.

[17] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *OASIS Workhop*, 2004.

[18] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. of the Workshop on Operating System and Architectural Support for the On- Demand IT Infrastructure*, Oct. 2004.

[19] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A new architecture for device drivers. In *Proc. of the Eleventh IEEE HOTOS*, 2007.

[20] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proc. of the 13th ACM ASPLOS*, Mar. 2008.

[21] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Proc. of the 2007 IEEE DSN*, June 2007.

[22] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced configuration and power interface specification, version 5.0. `www.acpi.info/spec.htm`, Dec. 2011.

[23] Intel Corporation. Accelerating high-speed networking with intel I/O acceleration technology. `http://download.intel.com/support/network/sb/98856.pdf`, 2006.

[24] R. Jones. Netperf: A network performance benchmark, version 2.1, 1995. Available at `http://www.netperf.org`.

[25] A. Kadav and M. M. Swift. Live migration of direct-access devices. In *ACM SIGOPS Operating Systems Review, 'Best papers from VEE and Best papers from WIOV', Volume 43, Issue 3.*, July 2009.

[26] A. Kadav and M. M. Swift. Understaning modern device drivers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 3-7 2012.

[27] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[28] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech.*, 20(5), 2005.

[29] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal Computer Science and Technology*, 20(5), Sept. 2005.

[30] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05. ACM, 2005.

[31] M. Mahalingam and R. Brunner. I/O Virtualization (IOV) For Dummies. `labs.vmware.com/download/80/`, 2007. VMworld 2007.

[32] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *SOSP*, 2011.

[33] P. Mochel. The Linux power management summit. `lwn.net/Articles/181888/`, 2006.

[34] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009.

[35] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. Revivei/o: Efficient handling of i/o in highly-available rollback-recovery servers. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*. IEEE, 2006.

[36] D. Narayanan and O. Hodson. Whole-system persistence. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 3-7 2012.

[37] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the 11th International Conference on Compiler Construction*, 2002.

[38] Z. Pan, Y. Dong, Y. Chen, L. Zhang, and Z. Zhang. Compsc: live migration with pass-through devices. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012.

[39] V. Paxson. Bro: a system for detecting network intruders in real-time. In *Proceedings of the 7th conference on USENIX Security Symposium*, 1998.

[40] PCI-SIG. I/O virtualization. `http://www.pcisig.com/specifications/iov/`, 2007.

[41] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: transparent checkpointing under unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, 1995.

[42] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[43] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proc. of the 29th Annual Intnl. Symp. on Computer Architecture*, May 2002.

[44] P. Ramachandran. *Detecting and Recovering from In-Core Hardware Faults Through Software Anomaly Treatment*. PhD thesis, University of Illinois, Urbana-Champaign, 2011.

[45] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39 – 47, may 2005.

[46] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *Proc. of the 21st ACM Symp. on Operating System Principles*, Oct. 2007.

[47] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proc. of the 200 EuroSys Conference*, Apr. 2009.

[48] P. Schmid and A. Roos. Hitachi's 4 tb hard drives take on the 3 tb competition. `www.tomshardware.com/reviews/4tb-3tb-hdd,3183-15.html`, Apr. 2012.

[49] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. *SIGOPS Operating Systems Review*, 30:213–228, 1996.

[50] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. In *Proceedings of the 8th Conference on File and Storage Technologies (FAST '10)*, February 2010.

[51] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proc. of the 6th USENIX OSDI*, 2004.

[52] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. of the 19th ACM SOSP*, Oct. 2003.

[53] Úlfar Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: software guards for system address spaces. In *Proc. of the 7th USENIX OSDI*, 2006.

[54] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM SOSP*, Dec. 1993.

[55] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. of the 8th USENIX OSDI*, 2008.

[56] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the 7th USENIX OSDI*, 2006.