

# CONSISTENT DEVICE COMMUNICATION IN RESTARTABLE TRANSACTIONAL DISTRIBUTED MEMORY SYSTEMS

*Steffen Gerhold, Christian Himpel, Alexander Weggerle, Thilo Schmitt, Peter Schulthess*

Institute of Distributed Systems  
Ulm University  
Ulm, Germany  
{firstname.lastname@uni-ulm.de}

## ABSTRACT

Programming conventional hardware devices using transaction-based drivers poses new challenges to Software Transactional Memory (STM) systems. This paper analyzes the interactions between transactionally executed drivers and non-transactional hardware devices using the distributed transactional cluster operating system Rainbow OS. We demonstrate solutions to guarantee device state and data consistency in case of aborts of driver transactions due to synchronization issues as well as after node failures. Additionally we present a driver framework which verifies the feasibility of our approach.

**Index Terms**— Fault tolerance, software transactional memory, device drivers, distributed computing

## 1. INTRODUCTION

Software Transactional Memory (STM) is an increasingly popular approach of concurrent programming. Locking protocols are no longer needed and negative consequences such as priority inversion and deadlocks are avoided simplifying the parallel programming effort. As a rule transactional programming will easily handle software synchronization issues, but creates the new challenge of building transactional drivers for non-transactional hardware devices. This paper describes a strategy for cooperation between transactional device drivers and non-transactional hardware devices in Rainbow OS, a fault-tolerant distributed cluster operating system. It then develops a framework for transactional device drivers which simplifies the task of writing new drivers.

The rest of this paper is organized as follows: subsections 1.1 and 1.2 present the distributed operating system Rainbow OS which serves as platform for the development of transactionally consistent device drivers. Section 2 presents detailed information on the challenges of programming non-transactional devices in a transactionally consistent environ-

ment and presents possible solutions. Section 3 classifies device types with regard to the additional complexity added by the transactional checkpointing facility. Section 4 gives an overview of a framework which encapsulates the mechanisms needed to make a PCI device compatible with the transactional environment. Section 5 describes future work and concludes.

### 1.1. Rainbow OS

Rainbow OS is a lean distributed operating running on a cluster of AMD64-compatible commodity PCs connected by Gigabit Ethernet LAN [1] [2]. It is written in Java and relies on the Small Java Compiler (SJC) [3] to translate the Java source text into native machine code. Rainbow OS directly runs on the PC hardware with no virtual machine involved. It provides the abstraction of a Transactional Distributed Memory (TDM) making any object within the TDM transparently and directly accessible from all participating cluster nodes. All read and write operations to a TDM object are executed within transactions and are subject to the optimistic synchronization scheme of *transactional consistency*. This consistency model creates shadow copies of objects which are about to be modified and thus gains the ability to undo all modifications made by a running transaction. If an access conflict occurs the violated transaction is aborted, its changes to the TDM are undone and it is started again.

Due to its lean design and its intuitive distributed programming model Rainbow OS is used as example and demonstration platform in computer science lectures.

### 1.2. Page Server Checkpointing Facility

To add fault tolerance to Rainbow OS, a checkpointing facility called *page server* stores checkpoints of the TDM on a persistent medium. Traditional distributed checkpointing approaches must explicitly guarantee the consistency of the assembled checkpoint; otherwise they might risk to suffer from the domino effect which renders the stored checkpoints useless. The page server within Rainbow OS elegantly avoids the

---

The work reported in this paper was supported in part by the German Research Foundation (DFG) under grant SCHU 1035/7-1

domino issues by taking advantage of transactional consistency. A special transaction running on the page server reads the TDM and stores all modified areas on a solid state disk drive (incremental checkpointing). The transactional consistency guarantees that the written checkpoint is internally consistent without incurring additional and explicit coordination overhead [4] [5] [6] [7].

## 2. CONSISTENT DEVICE COMMUNICATION

A hardware device can usually adopt one of many different configurations. Even simple devices such as a serial line controller offer a multitude of configuration parameters, i.e. baud rate, number of stop bits or parity type. It is crucial for the device driver to maintain up-to-date information about the current state of the corresponding device so that the driver can issue correct commands to the device. If the driver's perspective of the device state differs from the real current state of the device, the driver might issue commands which are incorrect or even destructive with respect to the device's real internal state. In common operating systems a specific hardware driver usually is the only system component which modifies the device state directly. All other system layers accessing the device's services issue their request to the appropriate device driver. Thus the internal information of the driver always matches the real state of the device. A typical sequence of device driver actions in a non-transactional operating system consists of the following steps: first, the need for a device access (i.e. a write access to a hard disk) arises; the driver then issues a specific request (i.e. an ATA write sector command) to the corresponding device. After the request has been handled by the device, it offers some kind of feedback (i.e. interrupt) to the driver to signal success or failure. Programming hardware drivers within a distributed transactionally consistent environment such as Rainbow OS poses an interesting challenge to developers since a driver task must satisfy additional constraints to ensure system-wide consistency. Unlike the Rainbow operating system the underlying PC hardware is not aware of transactional consistency and of the incidental transaction restart and fallback actions. There are only few hardware projects such as Suns (presumably discontinued) Rock processor which attempt to offer in hardware some transactional mechanisms to the software running on top of them, and no concepts have yet found their way into commodity PC hardware. There are two scenarios which require special handling: the abort and rollback of the transactions which encapsulates the current execution of the driver task and the restart of the Rainbow OS cluster from a previously recorded checkpoint.

### 2.1. Handling Transactional Abort of a Driver Task

According to subsection 1.1 all Rainbow OS code is subject to transactional consistency, so a transaction can be aborted

at any time due to synchronization issues. Since the device's internal state is unaffected by a transaction abort, inconsistencies between the driver's perspective and the real device state might arise. This can lead to undesired results, data loss or even a system crash due to incorrect device requests issued by the driver.

There are two alternatives to circumvent the inconsistencies mentioned above. The first solution aims at preventing the transactional abort and the associated inconsistencies between device and driver. The implied serialization of all commits however severely reduces overall performance and cannot be considered an adequate solution. The second alternative uses local memory which is not subject to transactional consistency as buffer between the TDM and the device. This technique called "Smart Buffer" [8] allows for safe and elegant communication with the device as it prevents data loss in case of aborts for both read and write access.

### 2.2. Handling System Restart from Checkpoint

In addition to transactional aborts a Rainbow OS driver must also be capable to handle system restarts correctly. As pointed out in subsection 1.2 in the case of a failure the Rainbow cluster might need to fall back to the last valid checkpoint at any moment. As a worst case full restart of the cluster nodes might be required. To cope with this all driver objects and their respective machine code reside within the TDM, so the current state of the drivers is stored in the checkpoint and restored during the fallback. Since the devices themselves are reset in the worst case, they might lose their internal state at least partially or restart to a default state. This can lead to discrepancies between the configuration stored in the TDM driver object and the real configuration of the device.

Traditional operating systems such as Linux and Windows face a similar problem when switching to hibernation mode. The contents of main memory are written to disk before the PC is finally powered off. When it is started again, the devices have lost their original configuration and the drivers are responsible to reconfigure them to the state present before the hibernation took place. These traditional operating systems typically notify the drivers of the upcoming switch to hibernation mode so they can save their configuration to non-volatile storage. In contrast to hibernation Rainbow OS does not notify the drivers that a checkpoint is being created as this process is designed to be as transparent as possible and the checkpointing frequency would be compromised by the need to wait for the necessary disk access to finish. Every time the driver modifies an important configuration aspect of its associated device it stores the new state within the TDM. The exact location of the configuration information is irrelevant as long as it resides within the TDM. Thus the current configuration of the device is mirrored in a TDM object and is automatically recorded by the checkpointing facility. Each checkpoint now contains configuration information on all de-

vices which were configured at the time of the last checkpoint. After a restart of a cluster node has been performed, all drivers are notified of that event and reconfigure their corresponding device to the respective states stored in the TDM. The reconfiguration is performed concurrently to other activities running in the Rainbow cluster. Directly after the cluster restart all devices are marked as busy so parallelly running tasks cannot access the unconfigured devices. If some task requires access to a device, it must wait until the device is configured and ready. As restarts are assumed to happen rarely, a small delay directly after the restart can be tolerated. It is more important to minimize the overhead of storing the current device configuration in the checkpoint.

### 3. CLASSIFICATION OF RESTARTABLE DEVICES

After the restart of a Rainbow cluster node all active devices are reconfigured to match the last stored configuration (see section 2.2). The amount of work involved to reconfigure the device depends mainly on two factors: the complexity of the internal device configuration and the complexity of the communication interface between the driver and the device.

#### 3.1. Complexity of Device Configuration

Hardware devices adopt many different configurations and switch between operational states while performing their specific work. Computation tasks might use a specific device and require a special configuration state of the device to be active (e.g. serial line baud rate, vga screen resolution, network card addresses, ...). These *mandatory* requirements typically outlive a node restart since the tasks are part of the TDM and are therefore restored from the last checkpoint (see subsection 1.2). As consequence the device is reconfigured to match the preserved configuration in the checkpoint; otherwise inconsistencies might arise. As an example consider a graphics card running in textmode. If the driver erroneously assumes the graphics card to run in graphics mode and issues graphics mode commands, the graphics system is likely to crash. Configurations which are allowed to be different before and after a restart are called *non-critical*; these include interrupt line, memory addresses for dma buffers or descriptors and those configuration properties which are not visible to other operating system components.

The number of critical configuration properties heavily influences the device reconfiguration complexity after a node restart. The more critical configuration properties need to be taken into account, the more work is required during reconfiguration. The non-critical configuration properties do not increase the reconfiguration complexity as they must be set on each node start regardless of it being a fresh start or a restart.

#### 3.2. Complexity of Device Interface

The software driver and its corresponding hardware device usually communicate through I/O ports or through memory mapped I/O. Depending on the type of device this communication interface can be rather complex. The more flexible an interface is the more complex the process of determining the correct I/O locations can become. Communication interfaces can typically be attributed to one of the following three basic complexity levels:

- *Fixed I/O*: Only few fixed I/O ports or I/O memory locations are used. As these locations are not changeable, the complexity to determine the correct locations after a node restart is very low; it is only necessary to check if the device is still available. Simple devices such as the legacy timer or the legacy keyboard are examples for this complexity class.
- *Variable I/O*: The I/O area related to a device can be changed by software, e.g. the BIOS or the operating system. Adding or removing a specific device after the cluster node has shut down and before it starts up again can therefore change the effective I/O locations of other devices. In addition to checking the existence of its device, the corresponding driver must also be able to detect changes in the I/O locations and to react appropriately. Examples for this complexity class include nearly all PCI and PCI Express devices as they are required to support flexible I/O locations by the PCI standard.
- *Hot Plug*: Communication interfaces which support adding and removing the device at runtime. The most prominent example of this group are USB devices such as an USB mouse or a USB pen drive (note that with respect to this categorization the USB host controller itself is not a *hot plug* device but a *Variable I/O* device). Communication between these devices and their drivers is usually not performed via hardware I/O locations but with logical channels provided by the host controller. After a node restart, the host controller must again enumerate the attached devices and assign logical IDs to them. Thus a specific device might be assigned an ID different from the one it possessed before the restart; additionally some device might have been unplugged and reattached to a different host controller when the system was powered off. It is crucial that the device driver is able to cope with altering logical IDs as well as with a change of the corresponding host controller.

The flexibility of the device interface is the second influence on the overall reconfiguration complexity of a device. Figure 1 illustrates both the interface and the configuration restart complexity for selected devices. The overall complexity of reconfiguring a specific device on a node restart increases as the device moves from the bottom left to the upper

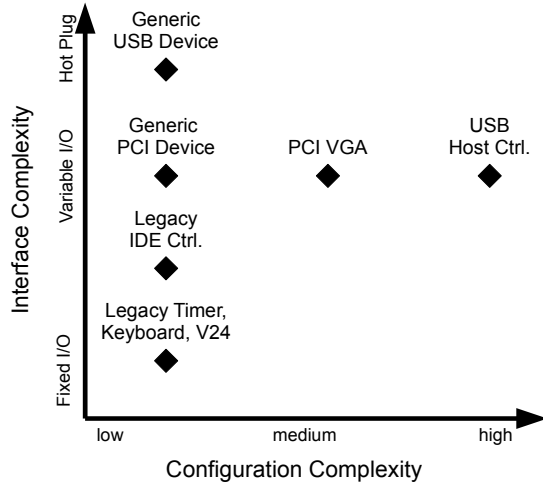


Fig. 1. Reconfiguration Complexity for Selected Devices

right corner of the figure. A legacy keyboard driver for example implies a low overall complexity whereas a USB host controller driver introduces a quite high reconfiguration complexity.

Based on this classification of restart complexity, groups of devices with similar complexity can be selected so the required reconfiguration functionality can be implemented in a generic way.

#### 4. FRAMEWORK FOR PCI-BASED RESTARTABLE DRIVERS

As described in section 2, drivers in Rainbow OS must correctly handle node restarts from checkpoints. It would be redundant to have every device driver implement its own reconfiguration code; thus we created a framework for the most common device types - PCI devices. Figure 2 shows the process of determining the appropriate driver instance for a PCI device detected during the restart. All information about a specific PCI device such as vendor ID, device ID or class code are aggregated and used to detect a preexisting driver instance which has been responsible for this device before the restart occurred and which is currently registered in the system-wide name service. If the appropriate driver instance is found, it probes the detected device and on success takes control of it. A special task is spawned which invokes the driver methods responsible for reconfiguring the hardware device according to the stored information. Should no adequate driver instance be available, the framework searches for a suitable driver class for the detected PCI device. If such a driver class exists, a new driver object is instantiated and registered in the name service. After that, a new task is created which takes care of the configuration of the device by calling special configuration methods of the driver instance. If no appropriate driver class can be found, the newly detected PCI device is unsupported.

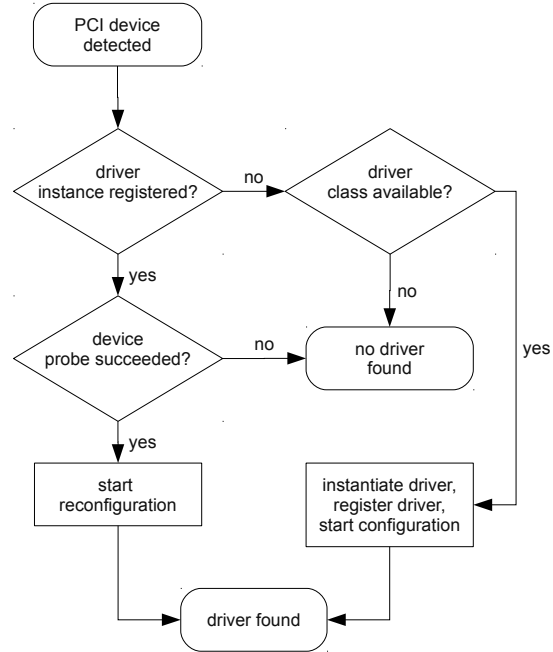


Fig. 2. Framework for PCI

By providing the functionality described above, the PCI framework greatly simplifies the implementation of a new PCI driver. All PCI-specific driver detection and reconfiguration-related code is accumulated within a Java base class called *RestartablePciDevice*, so a driver programmer can derive a new PCI driver class from this base class and inherit the predefined reconfiguration code. If special behavior is required, the methods of the base class can be overridden to offer specifically tailored functionality.

#### 5. CONCLUSION

We have shown an efficient way of controlling non-transactional hardware devices with transactional device drivers. Smart Buffers are used to decouple the transactional driver from its non-transactional counterpart device. A classification of devices into groups enables the implementation of generic utility classes which combine most of the functionality needed to augment the drivers with fault tolerance against node failures. The implementation of a fault tolerance framework for PCI drivers demonstrates the feasibility of this approach.

In the future we plan to extend the framework from PCI devices to generic USB devices to simplify the implementation of USB device drivers and increase the number of USB devices supported by Rainbow OS. Another approach showing promise is the development of hardware with active transaction support which shifts the complexity of transactional behavior from the driver to the hardware device. An easy way to gain first experience lies in modifying the hardware device emulations in virtualization solutions such as Qemu

and KVM and to improve the interaction between the emulated devices and the corresponding drivers in rainbow OS.

## 6. REFERENCES

- [1] N. Kaemmer, S. Gerhold, P. Schmidt, M. Sonnenfroh, S. Frenz, and P. Schulthess, "Transactional distributed 64 bit memory for pc clusters," in *Journal on the research topic "Eingebettete und Selbstorganisierende Systeme" of the 11. Chemnitzer Linux-Tage, Chemnitz, Germany, 2009*, 2009.
- [2] N. Kaemmer, S. Gerhold, T. Baeuerle, and P. Schulthess, "Transactional distributed memory management for cluster operating systems," in *Proceedings of the 32. International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 2009*, 2009.
- [3] S. Frenz, "Small java compiler," <http://www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/compiler.html>, 2009.
- [4] S. Gerhold, M. Schoettner, M. Fakler, M. Sonnenfroh, and P. Schulthess, "Smart snapshots on top of a distributed transactional memory," in *Proceedings of the Eurosys 2007, Lisbon, Portugal, 2007 (Poster Session)*, 03 2007.
- [5] S. Gerhold, P. Schmidt, A. Weggerle, and P. Schulthess, "Improved checkpoint / restart using solid state disk drives," in *Proceedings of the 32. International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 2009*, 2009.
- [6] S. Frenz, *Zuverlaessiger verteilter Speicher mit transaktionaler Konsistenz*, Ph.D. thesis, University of Ulm, 2006.
- [7] M. Schoettner, S. Frenz, R. Goeckelmann, and P. Schulthess, "Fault tolerance in a dsm cluster operating system," in *Workshop on "Dependability and Fault Tolerance", within the Int. Conference on Architecture of Computing Systems, Augsburg, Germany, 2004*.
- [8] T. Bindhammer, R. Goeckelmann, O. Marquardt, Schoettner M., M. Wende, and P. Schulthess, "Device programming in a transactional dsm operating system," in *Proceedings of the Asia-Pacific Computer Systems Architecture Conference, 2002*.