

# Pageserver: High-Performance SSD-Based Checkpointing of Transactional Distributed Memory

Steffen Gerhold, Nico Kaemmer, Alexander Weggerle, Christian Himpel, Peter Schulthess  
Institute of Distributed Systems  
Ulm University  
Ulm, Germany  
{firstname.lastname}@uni-ulm.de

**Abstract**—The Rainbow cluster operating system provides fault tolerance by storing incremental checkpoints on a solid state disk (SSD) drive. As the access characteristics of SSD drives differ substantially from those of traditional magnetic hard disk drives (HDDs) new interesting techniques of storing checkpoints in an elegant object-oriented fashion can be applied. The usage of transactional distributed memory (TDM) simplifies the distributed checkpointing algorithm as it automatically guarantees consistent checkpoints of the entire Rainbow cluster without additional coordination overhead.

## I. INTRODUCTION

More recently substantial research on software transactional memory (STM) [1] [2] [3] has emerged. Originally proposed by Shavit and Touitou [4] STM aims at simplifying parallel programming on multi-core systems by avoiding locks and consequential correctness issues such as deadlocks and priority inversion. In contrast to locking techniques, STM uses an optimistic concurrency approach and assumes the lack of conflict between parallelly running activities in most cases. Read and write accesses to shared memory issued by one specific task are bundled into a transaction which is atomically executed in virtual isolation. If two concurrent transactions perform conflicting memory accesses, only one of the two is allowed to continue; the other one is transparently aborted and all its modifications are undone.

### A. Rainbow OS

Rainbow OS is a lean cluster operating system [5] running on commodity AMD64-compatible PCs connected by a Gigabit LAN. It features a transactional distributed memory (TDM) which allows easy and transparent access across different cluster nodes to transactional objects independent from their physical location [6]. Access from all nodes to the TDM is synchronized by transactional consistency for all cluster operations [7] [8]. Rainbow OS is written in a subset of the Java programming language; The Small Java Compiler [9] transforms the source code into type-safe, object-oriented

native machine code. Because of its lean design and its paradigm of simplified distributed programming it is also used as example and demonstration platform in computer science lectures.

In contrast to thread-based STM systems such as DSTM [10] and FSTM [11] Rainbow OS uses virtual memory protection mechanisms to detect conflicts between concurrent activities on one or more cluster nodes. An optimistic synchronization approach creates shadow copies of data which is about to be modified and therefore gains the ability to undo all modifications performed by the currently running transaction in case of conflicts. A committing transaction notifies other cluster nodes of the data areas it modified while it was running and causes them to be invalidated on the other cluster nodes.

### B. The “Page Server” Checkpointing Facility

Fault tolerance is added by introducing a checkpointing facility named “page server” which saves incremental snapshots on a persistent storage medium. In case of node failures the entire Rainbow cluster can perform a fallback and restart from the last stored checkpoint. Common distributed checkpointing algorithms must pay special attention to create consistent snapshots and to avoid the domino effect. The page server can circumvent this complication in an elegant way: as the entire TDM is subject to transactional consistency, it is sufficient to have a transaction read all modified data and store it on a persistent medium. Earlier versions of the page server in the predecessor research operating system Plurix [12] [13] used magnetic hard disk drives (HDD) as storage medium. To achieve high disk write throughput and thus a high checkpointing frequency it was essential to write many consecutive hard drive sectors at once to avoid hard disk seek time penalties. This led to the development of sophisticated disk write strategies such as “linear segment” [14] which guaranteed high disk throughput by writing in a strictly sequential way. Although these write strategies allowed fast disk access, they also required complex disk defragmentation processes to ensure undiminished disk throughput in case of a nearly completely filled disk.

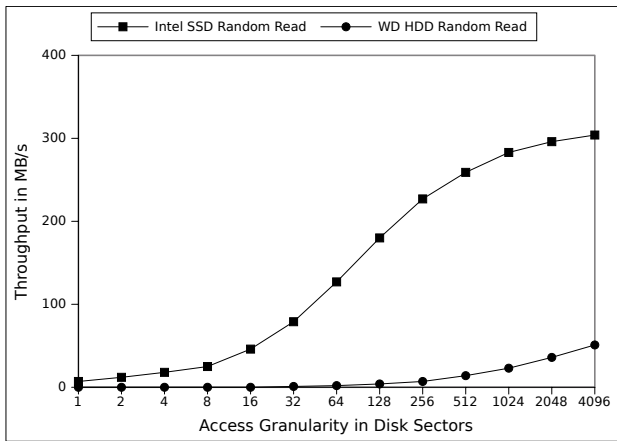


Fig. 1. Random Read Throughput

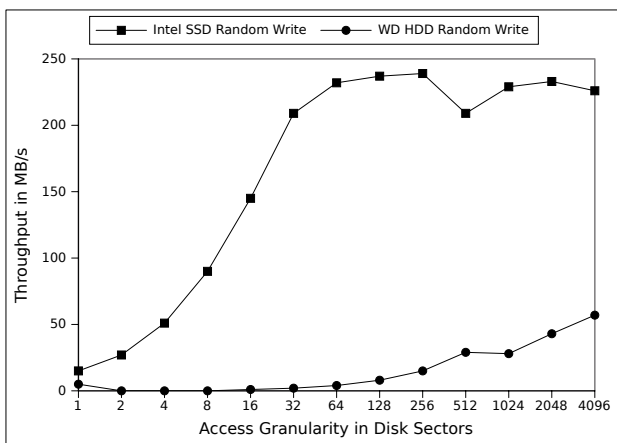


Fig. 2. Random Write Throughput

The advent of fast solid state disk (SSD) drives offers new possibilities for advanced disk write techniques which improve on the linear segment strategy on magnetic hard drives both in terms of disk throughput and in reorganisation simplicity and elegance.

The remaining paper is organised as follows: in section II an overview of SDD access characteristics compared to those of HDDs is given. Section III describes the pageserver checkpointing facility. Section IV describes possibilities for future work.

## II. SDD CHARACTERISTICS

As SSD drives grow in storage capacity and fall in price they get more widespread in the consumer market promising to speed up file access by providing low latencies and high throughput rates. To get an overview of HDD and SSD performance some micro-benchmarks that were performed for two magnetic hard disk drives (a Samsung SpinPoint S250 / 250 GB and a Western Digital Caviar Green / 2 TB) as well as two solid state disk drives (OCZ CORE SSD / 64 GB and an Intel X-25E / 32 GB). Table I shows the results of the *seeker*

<sup>1</sup> benchmark. The access time and the seeks per second were measured by issuing 512-Byte read commands to randomly chosen disk sectors. Additionally the random access disk throughput shown in figure 1 and figure 2 was detected with the internal disk benchmarks of Rainbow OS. It is evident that the access latency of SDDs is up nearly orders of magnitude smaller than the latency of HDDs when reading hard disk blocks with a random access pattern. Independent of the access granularity the SSD drives perform much better than the HDDs. It is important to note that the random write throughput in figure 2 reaches its maximum at an access granularity of 256 sectors or 128 KB and even decreases slightly at larger access granularities. This special access characteristic is due to the internal flash memory used in SSD. As flash memory can only be deleted in larger quantities called erase blocks, each write operation issued to modify an area smaller than the erase block size introduces additional SSD-internal read and write cycles and lowers the disk performance.

Hard Disk	Random Access Time [ms]	Seeks/s
Samsung HDD	13.62	73
Western Digital HDD	18.01	55
OCZ SSD	0.43	2336
Intel SSD	0.11	9166

TABLE I  
COMPARISON OF HARD DISK ACCESS CHARACTERISTICS

## III. PAGE SERVER

The page server is responsible for creating and storing consistent checkpoints of the entire Rainbow cluster.

### A. Memory Mapped Storage Medium

The storage medium is used as raw data space since a file system might have negative effects on performance and timing predictability. The paging mechanism is used to create a mapping of the storage medium in a small area of the 64 bit<sup>2</sup> virtual memory address space, similar to the *mmap* system call in Unix-based operating systems. Thus the pageserver is relieved from explicitly managing the hard disk sectors. On the one hand, hard disk reads can be performed by merely accessing the appropriate memory address within the virtual disk mapping. On the other hand, all data written to the memory mapped area are eventually stored on the corresponding disk sector. A certain amount of physical memory pages can be dedicated to serve as cache and to be mapped to those virtual memory pages which currently contain data read from disk or data that is to be written to the disk in the near future. Due to the principle of locality many accesses to the virtually mapped disk space can be performed generating only few I/O requests to the device. Access to virtual memory pages which have not yet been mapped leads to a page fault and a subsequent load operation from disk. Flushing the cache writes all modified

<sup>1</sup>[http://www.linuxinsight.com/how\\_fast\\_is\\_your\\_disk.html](http://www.linuxinsight.com/how_fast_is_your_disk.html)

<sup>2</sup>Despite the notion only 48 Bit are supported by current AMD64 architectures

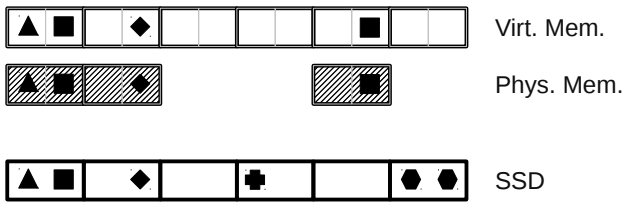


Fig. 3. SSD Mapping in Virtual Memory

data back to disk. To achieve high performance it is important to choose a good access granularity. A small granularity induces low access latencies in case of cache misses but a high page fault rate and a lower throughput rate. A large granularity reduces the number of page faults and increases the throughput but also causes higher latency. For SSDs the erase block size constitutes an effective lower boundary of access granularity, since write operations on smaller disk blocks cause performance degradation (see section II). Figure 3 demonstrates a simplified scenario in which operations on a memory-mapped SSD drive have been performed. Both the virtual memory and the SSD are subdivided into smaller fixed-size areas which match the erase block size of the underlying SSD storage medium. As the erase block size is usually larger than a single virtual memory page of 4KB, each fixed-size area in virtual memory is composed of several 4KB memory pages which are treated as whole by the page fault handler and the memory mapping routines. The geometric shapes signify data stored on disk or in memory. Depending on the access patterns this data can be identical in the disk block and the corresponding memory block or either of them might hold the most current version. If a freshly allocated block has not been written to disk yet, it might contain data which is not available on the hard disk block. If a virtual memory location within the disk mapping area has not been used since its last flush to disk, it might even lack a corresponding physical page and contain no data at all.

### B. Checkpointing Strategies

The address space of current AMD64-compatible PCs is huge and can contain up to 256 TB of data. It is futile to try and store exhaustive checkpoints of this vast area since the address space exceeds the available disk space by far. As only a small fraction of the address space is usually modified by transactions, it is reasonable to store incremental checkpoints which only include those data areas that have changed since the last checkpoint [15]. The only exception to that strategy is the first checkpoint stored for a newly started cluster which must contain all initially allocated TDM data areas.

The page server features two checkpointing strategies which differ in complexity and performance. Using *blocking checkpointing*, all currently running transactions in the cluster are deferred for a short time by disallowing them to commit their modifications. In the mean time a read transaction on the page server requests all modified data and stores it on hard disk. When the checkpoint is complete, all transactions regain their

```

public class MetaData {
    MetaData next;
    byte[] data;
    long tdmAddr, logTime;
    int crc, flags;
}

public class Checkpoint {
    Checkpoint next;
    MetaData metaData;
    long logTime;
    int flags;
}

public class Root {
    long magicId;
    Bitmap allocationBitmap;
    Checkpoint openCp, requestingCp,
    stableCp;
}

```

Fig. 4. Checkpoint Representation Classes

ability to commit. The blocking approach is the easiest way of storing checkpoints, but bears the disadvantage of blocking concurrently running transactions if these want to modify transactionally consistent data. *Concurrent checkpointing* removes the need to stall the cluster during checkpoint creation. The page server notifies all cluster nodes that it will now start to create a checkpoint. Consequently the nodes temporarily do not discard old shadow copies (see subsection I-A), but keep them as long as the checkpoint is being constructed. The page server requests the transactional image in the same way as he does during blocking checkpointing. For each incoming data request the cluster nodes check whether they keep shadow copies of the desired data and then answer the request by sending the old version of the data. If there are no shadow copies, the current version is sent. Thus the page server is guaranteed to receive a consistent image of the TDM without stalling the cluster. The fact that the checkpointed image is not completely up-to-date is negligible since the checkpointing frequency is sufficiently high to store a snapshot every few seconds. When the checkpointing process is completed, the page server once again notifies all cluster nodes which in turn dispose of the accumulated shadow copies.

### C. Data Representation

As the hard disk is mapped to memory it can be accessed just like any other memory location. This makes it possible to simplify data storage by creating Java-style objects within the memory mapped area. These objects can be used just like any other objects in Rainbow OS; they can be allocated with a call to *new()*, they can contain references to other objects and they can be safely reclaimed by a garbage collector. The following three object classes (see figure 4) are used to describe checkpoints:

- class *MetaData*: An instance of this class describes a memory area which is part of a checkpoint. It contains meta information such as the virtual address and the logical time as well as the content of the memory area encapsulated in a byte array. A next reference can provide a connection to the following *MetaData* instance in a linked list.
- class *Checkpoint*: A *Checkpoint* instance subsumes all *MetaData* instances which form an incremental checkpoint, i.e. the representations of all data areas which have been modified since the last checkpoint. In addition, the logical time of the checkpoint creation and a next reference can be included as well.
- class *Root*: Only one instance of the *Root* class exists. Its existence at the beginning of the memory-mapped disk area describes a valid Rainbow page server partition using a magic ID. Furthermore the root instance contains a *Bitmap* to store information on allocated disk blocks and references to different *Checkpoint* objects which are described in detail below.

Checkpoint creation consists of two phases. Firstly, all data invalidations since the last incremental checkpoint have to be recorded. Secondly, all invalidated data must be requested to form the new incremental checkpoint. Depending on the checkpointing strategy those two phases may overlap. It is thus not sufficient to store all currently incoming information into the same *Checkpoint* instance as the information might be relevant to a specific checkpoint only. For this reason three different references to *Checkpoint* instances with different purposes have been introduced in the *Root* instance:

- The *stable checkpoint (stableCp)* instance is the head of a linked list which contains all completed incremental checkpoints stored on the hard disk. When a new checkpoint has been created, it is added to this list.
- The *requesting checkpoint (requestingCp)* features a list of all data areas which have been invalidated since the last checkpoint. To create a new incremental checkpoint all these data areas have to be requested and stored on disk. If there are no more receivable data areas recorded in the requesting checkpoint, it has become a stable checkpoint and is added to the corresponding list.
- The *open checkpoint (openCp)* is used to collect all incoming information on data invalidation due to committing transactions. A new instance of the open checkpoint is allocated every time an incremental checkpoint is being started. It therefore contains the addresses of all data areas which have been modified since the last checkpoint. When a new incremental checkpoint is about to be created, the open checkpoint becomes the new requesting checkpoint and a new open checkpoint instance is allocated.

Figure 5 shows an example of a concurrent checkpointing situation: the requesting checkpoint holds a linked list of *MetaData* (*MD*) instances which partially have data objects with requested TDM data attached. The *MD* instance without

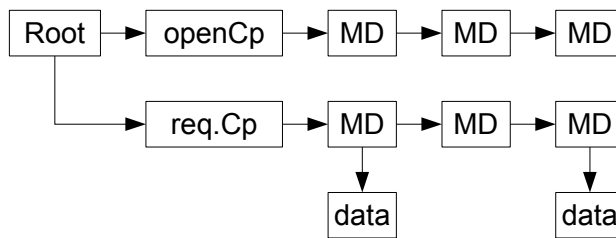


Fig. 5. Checkpoint Representation Example

reference to a data object is still waiting for requested TDM data to be received. As the cluster continues its operations during the checkpointing process, new data invalidations are being received and stored in the *MetaData* instances attached to the open checkpoint.

#### D. Reorganisation

To maintain high checkpointing performance reorganisation activities on the mapped disk area are essential. There are two main aspects which must be considered:

*Deletion of Checkpoints*: When incremental checkpoints are periodically created, the importance of a specific checkpoint decreases over time. A recent checkpoint is likely to be considered essential for reliability purposes, but there are few reasons to keep several older checkpoints which were taken one after another, because it is very unlikely that all of them will be eventually used. To free precious disk space it is advisable to delete unimportant checkpoints. However, simply deleting the instance of an unimportant checkpoint and all referenced meta data and attached data might lead to inconsistencies in subsequent checkpoints since all checkpoints are incremental and might depend on previous ones. To get rid of a checkpoint in a consistent fashion the page server needs only discard those data areas which are overwritten in the next checkpoint. All remaining data must be merged into the subsequent checkpoint.

*Defragmentation of Disk Space*: After some checkpoints have been deleted, the memory-mapped disk area is likely to be fragmented since the deallocation of unused objects provides small areas of unused space which are scattered throughout the disk area. It is beneficial to relocate objects which are still in use to create larger chunks of free memory of at least erase block size to reduce the frequency of page faults and thus avoid a decreasing checkpoint frequency. As all relevant information is represented as type-safe Java objects, all references to a specific object can be detected and a safe automatic relocation can be guaranteed.

## IV. FUTURE WORK

Several performance and functionality upgrades are considered for the near future:

- *Data Compression*: the TDM data sent throughout the cluster can be compressed to save valuable network capacity and to reduce access latencies. Here a tradeoff between compression ratio and the cpu time required to

compress the data is to be found. A reduced data size is also beneficial for the page server as it takes less disk space to store a specific checkpoint. The page server would not be involved in compressing or decompressing the data as a transparent storage of incoming cluster data is used. Any incoming compressed data will be directly stored using the compressed representation.

- *Distributed Page Server*: the current page server implementation enhances the reliability of the Rainbow cluster as long as it does not fail itself. In future the page server will be distributed across different nodes so that the failure of some number of page server nodes will be tolerated.

## REFERENCES

- [1] V. J. Marathe and M. L. Scott, "A qualitative survey of modern software transactional memory systems." University of Rochester Computer Science Dept., Tech. Rep., 2004.
- [2] B. Saha, A. Adl-Tabatabai, R. Hudson, C. Minh, and B. Hertzberg, "Mcrst-stm: A high performance software transactional memory system for a multi-core runtime," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Manhattan, New York, USA, 2006*, sTM.
- [3] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang, "Enabling scalability and performance in a large scale cmp environment," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, 2007, pp. 73–86.
- [4] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, 1995.
- [5] "Homepage of rainbow os," <http://www.rainbow-os.net>.
- [6] 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.
- [7] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, and P. Schulthess, "Optimistic synchronization and transactional consistency," in *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, 21-24 May 2002, pp. 331–331.
- [8] M. Wende, "Kommunikationsmodell eines verteilten virtuellen speichers," Ph.D. dissertation, University of Ulm, 2003.
- [9] S. Frenz, "Small java compiler," <http://www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/compiler.html>, 2009.
- [10] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures," in *Software transactional memory for dynamic-sized data structures*, 2003.
- [11] K. Fraser, "Practical lock-freedom," Cambridge University Computer Laboratory, Tech. Rep., 2004.
- [12] 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*.
- [13] 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.
- [14] S. Frenz, "Zuverlaessiger verteilter speicher mit transaktionaler konsistenz," Ph.D. dissertation, University of Ulm, 2006.
- [15] 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.