Improved Checkpoint / Restart Using Solid State Disk Drives

S. Gerhold and P. Schmidt and A. Weggerle and P. Schulthess
Institute of Distributed Systems
Ulm University
James-Franck-Ring O27, 89069 Ulm, Germany
Phone: (49) 731-5024239 Fax: (49) 50-24142 E-mail: steffen.gerhold@uni-ulm.de

Abstract - Fault tolerance and reliability of distributed systems is often achieved through checkpoint / restart mechanisms. Checkpointing frequency and restart delay crucially depend on data throughput and access performance of the storage medium. In this paper we discuss the opportunity to achieve subsecond checkpointing frequencies and restart delays by substituting magnetic hard disk storage with solid state disks and adapting the access algorithms and storage patterns to the novel constraints imposed by solid state disks.

I. INTRODUCTION

This paper presents the checkpointing facility of Rainbow OS and discusses the choice of checkpoint storage medium as well as the advantages of different access algorithms and storage patterns. The paper is arranged as follows: in section II a short outline of Rainbow OS is presented. Section III recapitulates several traditional checkpoint/restart approaches including related work and presents the transactional checkpointing fashion used in Rainbow OS. Measurements of disk performance and a discussion about the preferred storage medium are presented in section IV. Section V. evaluates different access algorithms and storage patterns before section VI. concludes this paper.

II. RAINBOW OS OVERVIEW

Rainbow OS is a distributed 64-bit operating system for AMD64-PC clusters connected via gigabit ethernet [1]. It is written in a Java-like language and runs directly on the underlying hardware without need for any Java Virtual Machine. By taking advantage of the features offered by the Small Java Compiler [2], Rainbow OS is implemented in a lean, object-oriented, type-safe way and supports multiple consistency models [3]. Objects are accessed directly in-memory without the need for middleware layers or wrapper classes.

Rainbow OS implements a page-granularity software transactional memory with support of the processor's memory management unit. All distributed objects and most parts of the Rainbow OS kernel reside within one big memory partition shared by all participating nodes. Accesses to this transactional distributed memory (TDM) are coordinated by a transactional consistency model [4] which allows for parallel access with optimistic synchronization: if several task try to modify the same memory location, i.e. an object variable, only one task wins and is allowed to publish its modifications. All modifications the other tasks were responsible for are being discarded and the tasks are restarted. This approach presents an intuitive single-station programming model to an application programmer by offering virtual isolation of each cluster node.

III. CHECKPOINT / RESTART

Checkpoint / restart is a common instrument to enhance reliability and fault tolerance in distributed systems. Time-consuming computations can gain much efficiency if they are checkpointed from time to time, as they can be restarted from the last checkpoint if any critical error occurs. If no such mechanism was provided, the computation would have to start over from the beginning thus losing all results achieved so far.

A. Traditional Checkpointing Approaches

Traditional approaches found in widespread operating systems such as Linux or Microsoft Windows typically focus on checkpointing applications, each of which might consist of one or several processes. There are three different techniques suitable for different checkpointing tasks [7]:

1) Application checkpointing: This scheme has the application itself bear all checkpointing responsibilities. As the internal structure and the specific importance of each data area is well known to the application programmer, a very efficient checkpointing with low overhead can be implemented. But there are several drawbacks to this solution as well: each applications needs to re-implement their specific checkpointing and restart routines which are obviously not transparent to the programmer. If the source code of some application is not available, there is no way to add a checkpointing facility to it. Furthermore there may be only few time-frames during the execution of the application allowing for a consistent checkpoint to be taken, so that the checkpointing frequency might get quite low.

2) Checkpointing libraries: To avoid re-implementing the checkpoint / restart facility for each application, checkpointing libraries have been developed. They can be linked to existing applications and provide independent checkpointing functionality. This approach is realized in frameworks such as libckpt [5] and cryopid [6]. As the library is part of the application's process, it can access all resources in the way the original application does. In comparison to application checkpointing the library can take checkpoints at any given time, being independent of time-frame constraints imposed by the application. A serious disadvantage of checkpointing libraries are enforced limitations in inter-process communication and

the use of system calls which render this approach unfeasible for parallel applications.

3) System-level checkpointing: The most powerful but also most complex approach moves the checkpointing facility from user space into kernel space. Not being confined by process boundaries it is now possible to checkpoint and restart applications completely transparently. In contrast to application checkpointing it is now necessary to save the entire process context as well, which includes the processor registers and the stack. Every time the kernel itself is changed by continuing development or security patches, a system-level checkpointing module might need to be updated; it therefore requires much maintenance. A well known example of this approach is Berkeley Lab Checkpoint/Restart [8]. Modern operating systems also offer a power saving mode called “hibernation” which is executed by creating a system-level checkpoint. It incorporates all running applications and additional kernel data, so that the complete state of the currently running machine is dumped to hard disk. From there it can be retrieved and used to quickly restart the operating system and put it back in the formerly saved state.

B. Distributed Checkpointing

Creating checkpoints of distributed applications is a non-trivial task. Uncoordinated checkpointing by each sub-application is not feasible as it may lead to domino effects rendering the stored checkpoint data useless. Different algorithms have been proposed to guarantee consistency of checkpoints taken in a distributed fashion. The approach of Chandy and Lamport [9] is easy to understand, but can not tolerate crashes of one or more participating nodes during checkpoint creation. More advanced algorithms such as the one of Koo and Toueg [10] are tolerant against node failure (they are said to be *resilient*), though they are much more complex and challenging to understand.

C. TDM Approach

The TDM checkpointing approach implemented in Rainbow OS has been developed in its predecessor system [11] [12] and facilitates checkpointing all distributed applications and the entire distributed kernel without introducing complex new algorithms. It guarantees a consistent image relying solely on existing TDM mechanisms and cannot be mislead by node failures.

As described in section II.A, a view on the entire TDM of Rainbow OS is always transactionally consistent. Running task work in virtual isolation which renders all modifications to the TDM visible at once in the moment of commit. Therefore a consistent image of the TDM can easily be obtained by starting a transactional task which reads the entire TDM and stores it on a persistent medium. In case of a node failure the cluster can be set back to the state stored within the checkpoint and continue running from there. In Rainbow OS a dedicated machine named pageserver is charged with the sole task of checkpointing the running cluster. This approach makes the checkpointing facility relatively immune against faults within the cluster, since the pageserver itself is not part of the TDM.

In order to reach a high checkpointing frequency and to save disk space it is not practicable to store the TDM as whole within each checkpoint written to disk. The amount of data to be included in a checkpoint can be greatly reduced by accounting only for those data areas which have changed after they have been written for the last time (incremental checkpointing)

IV. CHOICE OF STORAGE MEDIUM

In traditional operating systems, checkpoints are usually stored in files within a file system. Especially high disk throughput is often not deemed vital since checkpoints are not created frequently but only every few minutes. Rainbow OS is targeting to support a very high checkpointing frequency and does nor rely on file systems to store the checkpointed data. It uses raw access to hard disk drives to control the data structures on disk and to gain optimal disk performance.

A. Introduction to HDD / SSD

Earlier versions of Rainbow OS were using a specially developed algorithm called “linear segment” [11] to achieve high write throughput on magnetic hard disks. It is a well known fact that hard disk write performance depends crucially on minimizing the seek times between different write operations. By writing the (incremental) checkpointing data in a strictly linear fashion, the “linear segment” algorithm achieved very high write throughput rates on common magnetic hard disks.

The introduction of larger and cheaper solid state disks on the market now leads to the question whether common magnetic hard disks (HDD, hard disk drive) or solid state disks (SSD) are more suitable for providing persistent storage in the pageserver. The checkpointing facility can benefit from SSDs in at least two ways: firstly, SSDs show the promise of surpassing HDDs in both read and write throughput within the next years, as they integrate a continuously rising number of small flash memory controllers which can be accessed in parallel. Secondly, the access patterns and disc data structures of the current checkpointing implementation can be greatly simplified by leaving the paradigm of linear writing behind and switching to full write random access. As Rainbow OS is also targeted to be a teachable systems for university students to study, a minimization of complexity is surely desirable.

HDDs and SSDs use completely different approaches to store data and therefore require different access patterns for optimal performance. In theory HDDs perform well if the seek time is minimal and large contiguous data blocks are accessed. SSDs on the other hand typically show asymmetric read / write behavior. Due to their internal structure, read accesses are usually handled in an access granularity named “block size” which is usually few kilobytes in size. Write accesses however are processed in larger quantities named “erase blocks” which might span several hundred kilobytes (?) or a few megabytes (?). Whenever a quantity of data smaller than the an erase block is scheduled to be written, additional internal effort is needed. Those areas of the affected erase blocks which will not be updated by the data to be written must be read from the storage memory so that an entirely filled block

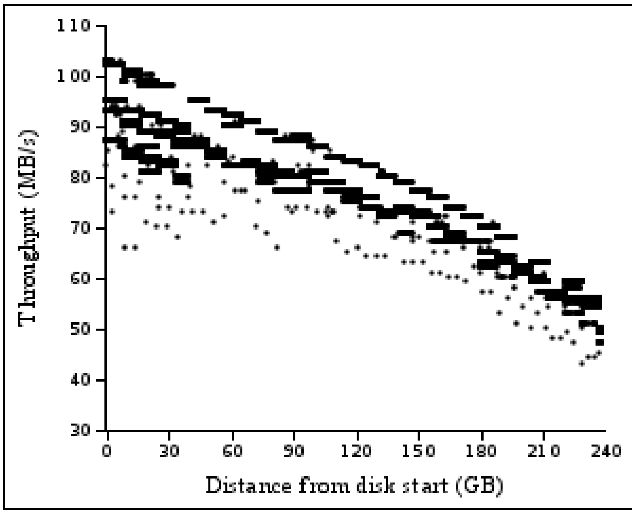

Figure 1: Read performance of Samsung HD250HJ magnetic hard disk in dependence of sector position

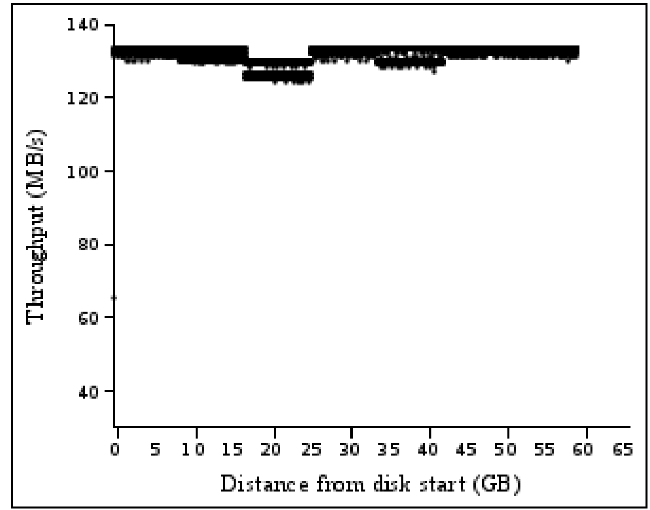

Figure 2: Read performance of OCZ CORE_SSD solid state disk in dependence of sector position

the size of an erase block is formed. This block can then be written to storage memory. This example clarifies the importance of writing in erase block granularity to SSDs in order to gain high performance.

B. Measurements

Theoretically, SSDs will provide many advantages when used as replacement for the currently installed HDDs in the pageserver. Measurements were conducted to test the feasibility of solid state disks as storage medium regarding throughput and constraints. The measurements were performed on a Gigabyte GA-D3MM-DS2R motherboard equipped with an Intel E8400 Core2 processor, 8 GB DDR2 RAM and an Intel ICH9R I/O hub which includes the disk controller. A Samsung HD250HJ disk with 250 GB, 7200 RPM, 8 MB Cache and SATA interface was selected as representative magnetic hard disk drive, whereas a OCZ CORE_SSD drive (OCZSSD2-1C64G) with 64 GB SATA II interface was the solid state disk of choice. All benchmarks used the IDE mode of the disk controller, as it was recommended by OCZ and the OCZ SSD only provided poor results in AHCI mode.

The first measurement aims at testing the disk performance not only at the beginning of the disk but spread over all existing sectors. For this purpose all sectors of the disks were accessed ascendingly with a granularity of 8192 blocks per access. The time it took for each access to complete was noted.

The results of accessing the HDD are shown in figure 1 (read access) and Figure 3 (write access). Both measurements indicate that the highest disk throughput will be achieved in the first part of the data area, whereas the speed declines with increasing sector number and reaches only half of its former value in the end. This behavior is typical for magnetic disk drives and follows from the internal storage design. The rotating magnetic discs within the HDD feature a higher speed in the outer ranges than in the inner ones which results in variable data throughput depending on the sector number. The average data transfer speed is therefore reduced to about 75% of the maximum throughput gained in the first sectors of the drive.

In contrast to the HDD the SSD shows no such decline in transfer speed. Figure 2 and figure 6 indicate a nearly

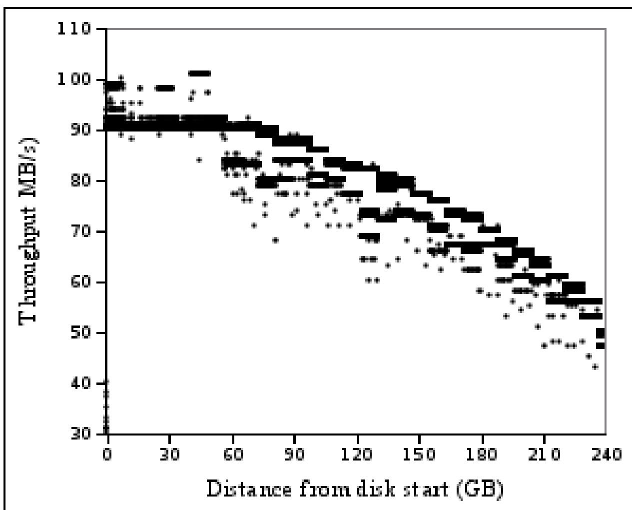

Figure 3: Write performance of Samsung HD250HJ magnetic hard disk in dependence of sector position

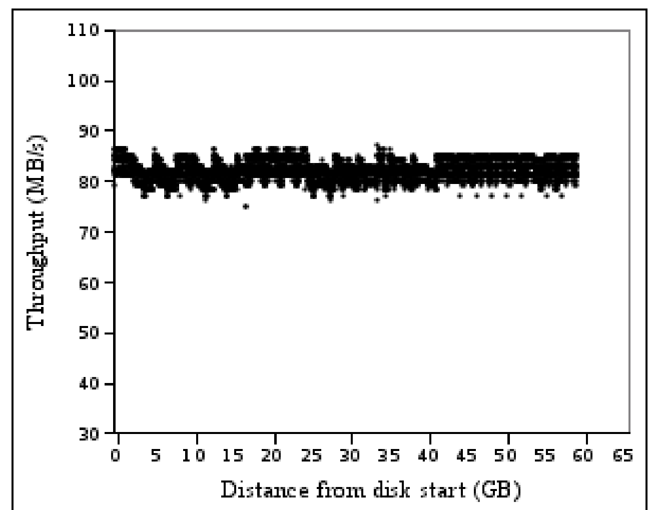

Figure 4: Write performance of OCZ CORE_SSD solid state disk in dependence of sector position

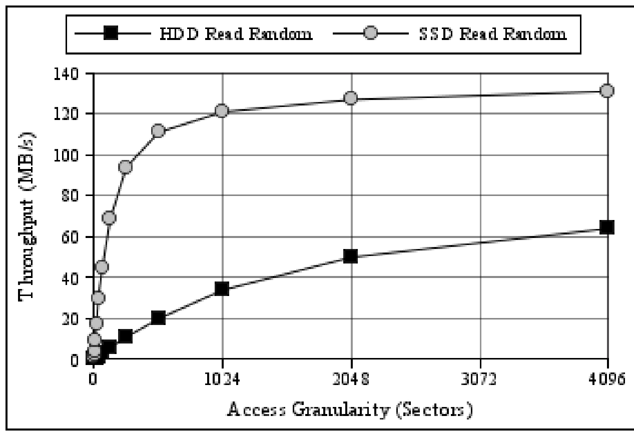

Figure 5: Random read throughput of HDD and SSD in dependence of access granularity

constant data throughput independent of the sector number. The SSD excels in read throughput with some 130 MB/s, but the average write throughput of about 85 MB/s can also compete with the HDD, as it is delivered constantly through all sectors of the disk.

Beside the ascending access performance it is also interesting to take a look at read operations of randomly chosen sector groups. If a restart is issued after a checkpoint, many Rainbow cluster nodes might request different data chunks from the pageserver. Small restart delays can be supported by a high random read performance of the pageserver. Figure 5 depicts a comparison of random read throughput on the HDD and the SSD in dependence of access granularity. As can be seen, the SSD provides better performance at all access granularities and approaches its read speed shown in figure 2 for larger sector groups. The HDD offers very low throughput especially for small sector groups.

The last measurement is aimed at comparing both disks assuming the usage of algorithms well adapted to the special constraints of each drive. A checkpointing facility tailor-made to use magnetic hard drives will have to write large groups of sectors in a linear fashion to achieve high performance. The throughput will be restricted by the highest linear write speed the HDD can offer. If the checkpointing facility is specially adapted to support SSDs it will rely on the relatively high write throughput an SSD can provide independent of the sector number. Its performance will therefore be limited by the maximum write random speed of its SSD. Figure 6 illustrates a comparison between the HDD write-ascending throughput and the SSD write random speed. It can be seen that the HDD provides a higher general performance than the SSD, but as this measurement was run within the first part of the sector area, the average speed of HDD write operations can be assumed to be 25% less, as depicted in figure 3. Taking into account this average speed, the linear write performance of the HDD and the random write throughput of the SDD are roughly equal, so both approaches can be considered as practicable and equally efficient.

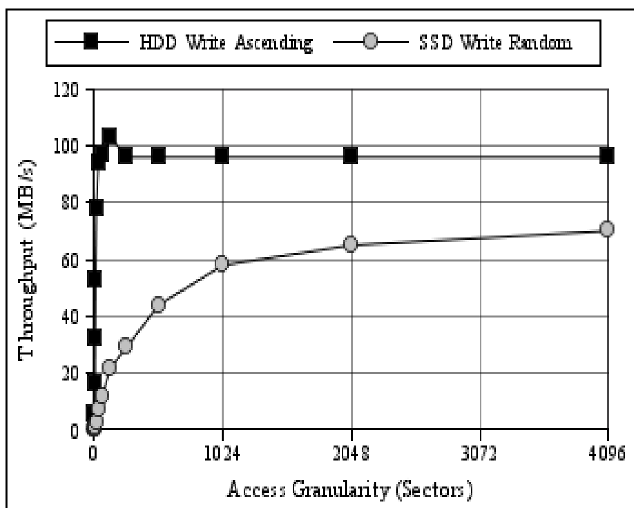

Figure 6: Write throughput of HDD (ascending) and SSD (random) in dependence of access granularity

C. Measurement conclusion

The measurements indicate that an SSD-bases checkpointing facility will be at least as efficient as a HDD-based one. The write performance is equal, even with respect to different optimal access strategies, so that no big differences are expected for checkpoint creation. Regarding the random read performance the SSD outperforms the HDD which leads to shorter restart delays in case of cluster failures.

Both disks used in this measurement are not cutting-edge technology. There are faster and bigger magnetic disks and solid state disks on the market. The last months showed tendencies of rapid development in both SSD size and throughput. Intel introduced its “X25-E Extreme” SSD which is supposed to achieve a write throughput of 170MB/s and a read throughput of 250 MB/s [13]. Samsung also announced a new SSD with 256 GB capacity promising it will reach a linear write speed of 200 MB/s [REF]. The current evolution indicates that in so far as speed is concerned, SSDs will leave HDDs very soon behind, then lacking only the large sizes of few Terabyte which current HDDs can offer.

Taking this development into consideration, the Rainbow checkpointing facility will rely on solid state disks to provide the persistent storage medium as they promise a reduction of complexity in disk structures and access algorithms as well as much higher throughput in the near future.

V. SSD AND MEMORY STRUCTURES

Earlier version of Rainbow and its predecessor systems used 32-bit PC cluster for operation. On these machines the size of the TDM was limited by the address space width of 32 bit to 4 GB. As mentioned in section II, Rainbow OS accesses the TDM in granularities of 4 KB pages. While a checkpoint is created all affected pages are read using a read transaction and stored on disk. To guarantee a correct restart from such a checkpoint, additional meta information must be stored into a checkpoint in addition to the raw page data. This includes the TDM address related to the page data, timestamps, flags and data checksums. The most important meta information part is the hard disk address of the corresponding TDM page, since only this information enables the pageserver to find the specific data of a TDM page by its TDM address.

If the TDM is limited to 2^{32} byte due to the use of a 32 bit machine, at most 2^{20} pages have to be taken into account, and the necessary meta information (2^{20} entries) can be handled and stored well. If one entry contains 16 byte, a total of 2^{24} byte (16 MB) is needed to store all meta information about one checkpoint in the worst case.

As Rainbow OS supports AMD64 machines, the available address space width increases to 48 bit[1]. In contrast to 32-bit machines 2^{48} entries are much more difficult to handle. If one assumes one entry to be 16 byte in size, the table of meta information would need 2^{40} byte (1 Terabyte) of memory. This consideration strongly hints that the 64-bit meta information concept cannot be carried over from the 32-bit approach. Not only is it advisable to have the checkpoint data being collected in an incremental fashion, but the meta information as well. In order to implement an efficient checkpointing facility based on solid state disk, several conditions should be met:

1) Access granularity: To ensure fast write throughput on the SSD, write operations should never transfer data blocks which are smaller than the SSD's erase block size. For larger SSDs this might implicate to collect many data pages and write them on disk using only one I/O operation.

2) Meta information size: the size of the meta information should be small enough to be stored in the pageserver RAM. If meta information must also be written to or read from disk during checkpointing activities, this might slow down the checkpointing process unnecessarily. To guarantee this condition even in worst case, a restriction of the TDM address space to less then 48 bit might be unavoidable.

3) Fast detection of meta information "delta": As considered above, it will be useful to store the meta information in an incremental fashion. The "delta" of meta information is defined as all those meta information entries which have changed since the last checkpoint. If this "delta" set cannot be computed very fast, e.g. because it requires many disk operations, the checkpointing process might be unduly prolonged.

There are many different possibilities to structure the meta information. Two promising possibilities, in-place arrays and hash tables, are being examined with regard to the above-mentioned criteria in the remaining part of this section:

1) In-place array: an in-place array consists of one meta data entry per available TDM page. Each entry of the meta information table can be easily found by shifting the address of the corresponding TDM page several bits to the right. As illustrated above, an array of this kind cannot be realized if the TDM address space is too large. By restricting the TDM address width to 40 bit, an in-place array will only require 4 Gigabyte of RAM. Clustering several data blocks to reach a sufficient access granularity can also be performed easily, as all dependent array entries can be found and modified very quickly. The detection of the meta information "delta" is also a very straightforward task: every time an entry is modified, a bit can be set in the entry itself or in a separate bitmap. This way it is instantly evident which entries have been changed since the last checkpoint. After a checkpoint has been created, all bits can also be reset very easily. It is also very convenient that the meta information table is implicitly sorted by TDM address.

2) Hash tables: arrays which are only sparsely filled are typically implemented as hash tables. This approach minimizes the necessary memory amount needed for the meta information in average case, but also introduces additional complexity. A hash table entry uses more memory space than its in-place array counterpart since the primary key by which the hash table is indexed needs to be stored as well to avoid ambiguities. Thus a hash table being able to include a fixed number of entries will be slightly larger than a comparable in-place array in the worst case.

Using a hash table it is unproblematic to aggregate several data pages into a chunk which is larger then the erase block size, as each entry can usually be found by computing the hash function of its address. The computation of the meta information "delta" can be simplified in the same way as it is done for the in-place array by marking modified entries with a special bit.

Both in-place array and hash table seem to be equally fit to be used as meta information structure. In average case the hash table needs less memory than the in-place array, but this advantage is inverted in a worst case situation.

VI. CONCLUSION

This paper gives a short introduction of the distributed Rainbow operating system and its checkpointing facility. It is elaborated how Rainbow OS simplifies the process of checkpointing the entire operating system by relying on existing distribution and consistency mechanisms. Subsequently the measurements of disk performances of magnetic hard disks and solid state disks are discussed and it is shown that SSDs provide an excellent basis for Rainbow OS to store checkpoints upon. In the end different disc and memory structures are proposed.

REFERENCES

[1] Homepage of Rainbow OS, http://www.rainbow-os.net

[2] S. Frenz, Homepage of Small Java Compiler, http://www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/compiler.html

[3] N. Kaemmer and S. Gerhold and P. Schmidt and M. Sonnenfroh and S. Frenz and P. Schulthess, "Transactional Distributed 64-Bit Memory for PC-Clusters", *Workshop on Innovative Operating System Concepts (WIOSC), Chemnitz Linux Days*, 2009, in press

[4] M. Wende, "Communcation model of a distributed virtual memory", Ph.D. thesis, Ulm University, 2003 (in German)

[5] J. S. Plank and M. Beck and G. Kingsley and K. Li, "Libckpt: Transparent Checkpointing under Unix", *Usenix Winter Technical Conference*, 1995

[6] CryoPID - A Process Freezer for Linux, http://cryopid.berlios.de/

[1]Despite the notion AMD64 systems currently support a maximum of 48 bits in virtual addresses.

[7] E. Roman, “A Survey of Checkpoint/Restart Implementations”, *Technical Report*, Berkeley Labs, 2002

[8] P. Hargrove and J. Duell, “Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters”, *Proceedings of SciDAC 2006*, 2006

[9] K. Mani Chandy and Leslie Lamport, “Distributed snapshots: determining global states of distributed systems”, *ACM Trans. Comput. Syst.*, vol. 3, p. 63-75, 1985.

[10] R. Koo and S. Toueg, “Checkpointing and Rollback-Recovery for Distributed Systems”, *IIEEE Transactions on Software Engineering*, vol. 13, p. 23-31, 1987

[11] S.Frenz, *Reliable distributed memory with transactional consistency,* Ph.D. thesis, Ulm University, 2006 (in German)

[12] S. Gerhold, M. Schoettner, M. Fakler, M. Sonnenfroh, P. Schulthess, “Smart Snapshots on top of a Distributed Transactional Memory”, *Proceedings of the Eurosys 2007 (poster session),* 2007

[13] Intel Corp., *Intel® X25-E SATA Solid State Drive*