Parallel Ray-Tracing with a Transactional DSM

S. Frenz, M. Schoettner, R. Goeckelmann, P. Schulthess
Ulm University, Distributed Systems Department
frenz@vs.informatik.uni-ulm.de

Abstract

Distributed Shared Memory (DSM) is a well-known alternative to explicit message passing and remote procedure call. Numerous DSM systems and consistency models have been proposed in the past. The Plurix project implements a DSM operating system (OS) storing data and code within the DSM. Our DSM storage features a new consistency model (transactional consistency) combining restartable transactions with an optimistic synchronization scheme instead of relying on a hard to use weak consistency model. In this paper we evaluate our system for the first time with a real parallel application, a parallel ray-tracer. The measurements show that our DSM scales quite well for this application even though we are using a strong consistency model.

Keywords: Distributed Shared Memory, Parallel Ray-Tracing, Operating Systems, Consistency Models.

1. Introduction

Commercial operation systems (OS) like Unix, Windows or MacOS use sockets or remote procedure calls for network communication. Traditionally, socket interfaces impose implementation of application specific protocols with oodles of error conditions upon the developer. As a response numerous middleware software packages like RMI, CORBA, and .NET offer a rich set of communication functionality, but fail to simplify implicit sharing of data. They provide distributed database functionality, object exchange, and messaging, but very limited consistency. Shared data has to be serialized for transport between disjoint address spaces, references have to be resolved, and often the full transitive closure for pointers has to be identified and handled before transmission. The resulting software system is dispersed across multiple software layers and development tools.

As an alterative, the possibility of sharing an address space offers an uniform view of the data. Distributed shared memory (DSM) systems proposed by L. Keedy [1] and K. Li [2] in 1985 and 1988 respectively, can implicitly handle access to shared objects without serialization nor resolving of references. Thus the developer gets a transparent view at shared data and may disregard aspects of distribution.

Equally important is the automatic consistency management of replicated data in the DSM. Many consistency models have been proposed by the DSM community [3]. Weak and weaker consistency models were introduced and became thus more efficient but also harder to program. Originally, DSM was developed to allow execution of parallel programs written for expensive multi-processor machines on cheap commodity clusters without major modifications. But using weak consistency models required artful modifications to the source code. We believe that this requirement is one of the reasons why the DSM concept is still not widely accepted, not even for new application fields.

Considering this defect the Plurix project implements the first OS storing data and code within the DSM. By storing everything within the DSM Plurix implements a real Single-System-Image (SSI). The latter is widely accepted within the cluster computing community and each user gains a global and uniform view on available resources and programs and it provides the same libraries and services on each node in the cluster, which is very important for load balancing and migration of processes.

Orthogonal persistence is another DSM property in the sense that any object reachable from the root of the cluster-wide name service can persist independent of its type. Persistence is directly supported by our checkpointing and recovery facilities and does away with de- and serialization functions required for file-based systems.

Because system data is also stored within the DSM we need a strong consistency model have therefore introduced the concept of transactional consistency into OS construction. The latter relies on restartable transactions and an optimistic synchronization scheme. Although we plan to introduce additional weaker consistency models in the future to support number crunching, the system executes fast even when using strict transactional consistency.

We expect that the Plurix system will open up new application fields for DSM systems like virtual worlds, telecooperation applications and multi-player games. But also traditional DSM applications are addressed like the parallel ray-tracing application explained in this paper.

The paper presents for the first time real performance data from an evaluation of our DSM-based Plurix OS with a real application and the system itself simultaneously in the DSM. In previous work we

presented a preliminary evaluation where synthetic memory access patterns were used and the system was stored outside the DSM.

In section 2 we present those parts of the Plurix system which are relevant for the performance evaluation. Subsequently, we discuss the architecture of our parallel ray-tracer and its parameters used for the measurements. In section 4 we present the data obtained from the performance evaluation. In section 5 we compare our results to related work. Finally, we give an outlook on future work.

2. The Plurix Operating System

The Plurix OS is inspired by Oberon system developed at the ETH Zurich by Wirth and Gutknecht [4]. The type-safe implementation language is essentially Java with some extensions. Hardware-independence is abandoned in favor of machine level language extensions and because the performance of the JVM is not sufficient. Therefore, we have developed a proprietary Java compiler directly translating java source texts into Intel machine instructions [5].

2.1 Memory Organization

Distribution is achieved in a page based distributed shared memory (DSM) presenting an identical view of a single distributed heap storage (DHS) to all nodes.

The transactional consistency model (see section 2.4 for details) is implemented for the DHS, accommodating both data and code, i.e. there is no separation between user- and kernel-space [6]. Only hardware management pages, space for non-transactional interrupt data (called interrupt-space) and local stacks are allocated outside the DHS. But this memory area can only be accessed via special functions of the compiler, available to kernel and device drivers only.

Since all nodes have the same view to DHS and all objects are located in DHS, sharing objects among nodes is easily done by using the cluster-wide name-service. Presence of an object is automatically distributed by entering it into the name service. This changes the DSM invalidating relevant pages on other nodes and yielding the desired object to the name service on any node at the next access.

To prevent indispensable classes and objects from being invalidated, these are located on special sys-pages. Examples are the memory management or the network driver. As a matter of course, all classes called by system-relevant classes in critical situations must also be system-classes.

2.2 Interrupt Handling

Interrupt handlers in Plurix as well as in other OSs require special care. The handler called by the hardware in case of a hardware interrupt have to be always present and has to reside on a sys-page. Since not all interrupt-service-routines are critical to the system, it is not mandatory to have all device drivers with service-routines and all therefrom called classes or objects stored on sys-pages.

Therefore Plurix implements a two-staged nested interrupted handling: the first stage detects the origin of the interrupt and then decides on the next procedure. Either the interrupt was requested by the network card, then the network device handler is called. Otherwise the first stage interrupt handler re-enables the network-interrupt and calls the second stage interrupt handler inside a device driver, which can be a regular object.

For interrupt driven I/Os non-transactional buffers are necessary which are described in [7].

2.3 Scheduler

Instead of having traditional processes and threads, the scheduler in Plurix works with transactions. We have adopted the cooperative multitasking model from the Oberon system. In each station there is a central loop (the scheduler) executing a number of registered transactions with different priorities. Any TA can register further transactions. System TAs, e.g. the garbage collector are automatically registered by the OS. Furthermore, the OS automatically encapsulates all user commands within a transaction.

2.4 Restartable Transactions

Any memory access to the DHS is encapsulated into transactions that follow the ACID (atomicity, consistency, isolation, durability) properties [8].

In Plurix durability is granted in an alleviated form taking into account the trade-off between topicality of persistent data and performance of commit. As all data and code resides in the DHS, saving the heap saves all relevant information. As all data is modified only within transactions, saving all changed pages saves the heap. To this end a pageserver continuously collects modified pages and writes them to disk [9]. In configurable intervals (for example: every two seconds) and whenever convenient the pageserver finalizes a consistent image. In case of error, fault, drop out or even after shutdown this image can be used to restore the cluster with consistent data.

If two or more transactions collide, at least one is aborted. The optimistic assumption is that unrelated

transactions will rarely conflict with each other. Conflicts are checked at the end of each transaction according to a forward validation scheme [10], i.e. the page addresses of all modified objects of the committing transaction are compared against all accessed objects of other active transactions. If a conflict is determined, one or more transactions must be restarted. Currently, we use a first-wins strategy that will be extended to improve fairness.

2.5 Transactional Consistency

Transactional Consistency works for both object and page based distributed memory systems. To simplify matters here the page based mechanism is described, but variable- or object-granularity are analog.

Memory management must keep track of all accesses to shared pages. Every transaction starts with all memory pages marked as "not accessed" and "read only". Reading a page (or object) sets its state to "used + read only". Writing to an object triggers the creation of a backup copy of original page, which is needed again in case of abort or in case of external paging requests. The new state of this object, which is now visible only to the current transaction, is "used + written" and "readable + writable". An ending transaction publishes all page numbers that have been written within this transaction in a write-set message, so that each node can compare the list of locally used pages with the list of the published write-set to detect a possible collision. In case of a collision every affected node resets its changed pages and restarts the current transaction.

Comparison to Other Consistency Models

Major issues in distributed systems or multiprocessor environments are topicality and semantic correctness of shared memory. Topicality is controlled by the memory model, but semantic correctness usually is managed by the programmer.

The following pseudo-code example illustrates both issues: given two shared variables x and y describing a geometrical point, node 1 wants to square and node 2 wants to rotate the shared point. Both operations read x and y, need a temporary variable for calculation and store their results back to x and y.

Node 1	Node 2

shared float x;
shared float y;
void quad() {
  float a=x;
  x=x*x-y*y;
  y=2*a*y;
}
	shared float x;
	shared float y;
	void rotate() {
	  float b=y;
	  y=x;
	  x=-b;
	}

If node 1 changes x/y without notification of node 2 before node 2 will calculate another x/y, the calculation of node 1 is lost. If node 1 and node 2 "simultaneously" calculate a new pair x/y, the execution might be:

void quad() {	void rotate() {
float a=x_1;	float b=y_1;
x_2=x_1*x_1-y_1*y_1;	
	y_2=x_2;
y_3=2*a*y_2;	x_3=-b;
}	}

In this case x_3/y_3 is destroyed, as it is the result neither of quad nor of rotate. This means if memory is not strict consistent, both nodes may read "old" values and therefore destroy new results or also overwrite one another. So explicit synchronization by the programmer is needed.

Even with strict consistency, there is the danger of loosing calculation or totally invalidating information: even though each single access to memory is strictly consistent, for most operations a semantic group of accesses must start with strictly consistent data and must then run atomically, i.e. several semantic groups with write-access to the same shared values are executed sequentially.

Consequently most programming systems have introduced barriers or locks, whereby setting these constructs require the nodes to communicate with each other. In this example each method with access to x or y has to inspect a lock in case of reading and to atomically test-and-set a lock in case of writing.

Transactional consistency provides strict consistency and includes semantic aspects in the form of transactions, because writing to a shared variable forces serialization of affected transactions, but parallel reading is allowed with full performance and latest data.

Implementation of Protocol

In Plurix not every access to an object is tracked, but only the first access to a page. This reduces the tracking overhead and enhances performance with the assistance of processor-built-in memory management unit (MMU) of Pentium or above CPUs, which automatically let the hardware set the appropriate bits in the page management tables.

Reading from or writing to a page which is not present results in a page-fault, which in turn requires the kernel to fetch this page from another node. After receiving the missing page program execution proceeds as if the page had been already present before the page fault, i.e. in case of reading: setting the "used"-bit by MMU, and when writing: raising protection exception and creating a shadow page as described above.

In page-based systems like Plurix at the end of a transaction there are two options to inform other nodes of modified pages: either by sending the content of the pages (update protocol: pages on other nodes are automatically refreshed) or by sending the numbers of all modified

pages (invalidate protocol: pages on other nodes are discarded and refreshed on access). Plurix implements the invalidate-semantic, as this offers reduced network load if not all pages are required by other nodes.

To ensure atomicity (i.e. sequential order) of separate end-of-transaction operations, Plurix uses a token-mechanism to arbitrate which node out of several competing ones may commit. Each commit increases the current 64-Bit commit-number, which can be used as cluster logical time. This commit number is included in every packet, so all receiving nodes can easily determine whether they missed a commit action and initiate appropriate recovery [9].

A node receiving a commit message will invalidate all locally mapped pages listed in this message. If the current TA used at least one of these pages it will restart as there was a collision with the currently committing transaction.
If a TA wants to revoke its changes or does not need its changes, it may opt to abort itself. An example for voluntary abort is a packet receiver which finds that there is currently no packet available. The voluntary abort will avoid imposing an unnecessary commit on the other nodes. For the DHS memory manager this voluntary abort is like a collision, so all written pages are restored from shadow images. This saves network bandwidth and commit processing time on this node (creating packet, sending packet) and on all other nodes (receiving packet, checking pages), and it possibly saves substantial amounts of memory and time for garbage-collection.

For example in case of an error-detecting compiler-run with hundreds of new objects, these objects are easily cleaned up with voluntary abort at minimized costs.
As described above Plurix currently uses a first wins strategy. If balanced fairness or prioritization of transactions is needed there are several implementation options.

Some handicapped or higher prioritized TA could signal invalidation of all needed pages and ignore the page-request of other nodes. As a consequence all nodes conflicting with this TA are forced to wait for this transaction to commit. This algorithm is not recommended as it is hard to control, delays many transactions and easily leads into deadlocks.

Another way to achieve fairness is to ask other nodes if they do not object to commit on modified pages. This leads to delay at end-of-transaction, but offers some choice on selecting the winning transaction and avoids deadlocks.

As there is no fairness problem with our current applications, there is no need of prioritization, but this will be of interest in future work.

Our DSM protocol is based upon non-reliable broadcast and non-reliable unicast in a single LAN-segment. Internal mechanisms check received messages and global time and lost packets will be detected. Action after packet loss depends on type of packet: most packets are re-requested and therefore a simple time-out can handle packet loss. Only packets belonging to a commit are not requested and therefore can not merely be re-requested.

The current implementation handles this severe error condition by falling back to a previously saved system image from the pageserver. The fall back operation takes less than 240 ms on Athlon XP+2000 until command prompt. The current implementation does not automatically restart old transactions after a recovery event as one of them could have caused the fallback.

3. Parallel Ray-Tracer Application

3.1 Implementation

To verify the performance of the transactionally consistent DSM, we ported the ray-tracer used by project 5 of class 6837 at Massachusetts Institute of Technology [11].

Supported objects of this lean ray-tracers are spheres and triangles. Each object may have its own surface, which describes color, shininess and the coefficients for the used phong model: ambient, diffuse and specular.

Available light sources are point lights, that feature adjustable intensity without direction and are mostly comparable to a bulb with zero size, and an ambient light, which is omnipresent and therefore raises no shadow.

For each display pixel a ray is initialized with a corresponding angle with respect to the camera. Each ray is intersected with all objects (no frustum) and restricted to the nearest one with intersection, giving the intersection point of this ray.

The color of an intersection point is calculated from the surface of the intersected object, the visible lights at this point and the reflection of the intersected ray. To calculate the reflection a new ray is initialized and traced, which results in recursive calls to the intersection method.

All rays are independently calculated but use the same scene-description, which in our case is shared via DSM and retrieved from the DSM-based name-service. Access to objects describing the scene is read-only, so there is no conflict. As successful write to a shared object leads to aborts on all machines which have at least read the modified page of this object, there would be an blatant bottleneck in calculation if several transactions try to publish their results via writing to a single page.

To retain usability and responsiveness for cooperative multitasking, not a complete block or even a single line should be calculated at once to retain usability and

responsiveness. In a number crunching environment this is not important, but here a ray-tracer is used to test transactional consistency and over-all performance of Plurix. So calculation of one line is divided into adjustable periods of calculation admitting intermittent user-inputs.

We have used a two-phase algorithm: a very short phase to allocate lines and a time consuming stage to calculate pixels in allocated lines. Compared to calculation of pixels, memory-allocation is not time consuming.

There is no static partitioning of the picture to be calculated, each node allocates a block of lines, calculates all pixels in this block and then proceeds with a new block, if there are any left. At the end of calculation there is a short period of time where only a few nodes are calculating, i.e. running time is not identical on all nodes.

3.2 Parameters of Ray-Tracer

There are several parameters affecting the time to calculate the same scene:

- pixels per line
- number of lines
- lines per allocation-block
- time per calculation-period
- number of nodes

Pixels per line and number of lines each affect time in the linear proportion. Lines per allocation-block determines number of allocations and therefore in combination with number of nodes is linked with probability of collisions at allocation of a block. Increasing lines per allocation-block on the one hand increases probability of collision because time to allocate grows, but on the other hand it decreases probability of collision because less blocks are allocated.

Time per calculation-block determines the total number of transaction-calls needed to calculate this block and therefore indirectly time consumed by operating system and other transactions.

The number of nodes should be inversely proportional to time in best case. In reality this is affected by administration effort, network traffic and collisions.

4. Performance Evaluation

The measurements were performed in a cluster with 12 nodes, each configured as following:

- processor: Athlon XP2500+ at regular 1.8 GHz
- motherboard: Asus A7V8X-X
- memory: 512 MB DDR-RAM
- network: 3com 905B-TX at 100 MBit half duplex connected to Allied Telesyn International CentreCOM MR912TX HUB

The scene used for ray-tracing is lit up by 3 light sources and consists of 8 triangles and 104 spheres with reflecting surfaces:

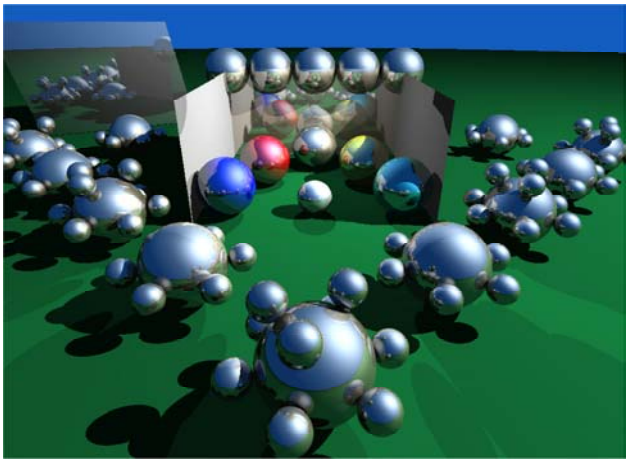

Figure 1. The Scene

Several test series with different parameters (see chapter 3.2) were made to test the performance of transactional consistency and to show the influence of varying the parameters. To cope with the plethora of measurements, diagrams are divided by aspects. The legend of a diagram contains resolution in pixel, lines per allocation-block and time per calculation-block. The longest time of all calculating nodes (see section 3.1) is taken for the diagrams. All times are in milliseconds.

4.1 Allocation-Block and Transaction-Time

All pictures are calculated with resolution 640x480, so the time spent usefully is identical. Changing the length of the calculation-block from 30 ms to 10 ms implies more transactions and therefore more overhead by the cooperative multitasking, as all other transactions in the central loop will be executed more frequently. Changing the allocation-block from 40 lines to 8 lines creates more collisions and is irrelevant if only one node is calculating.

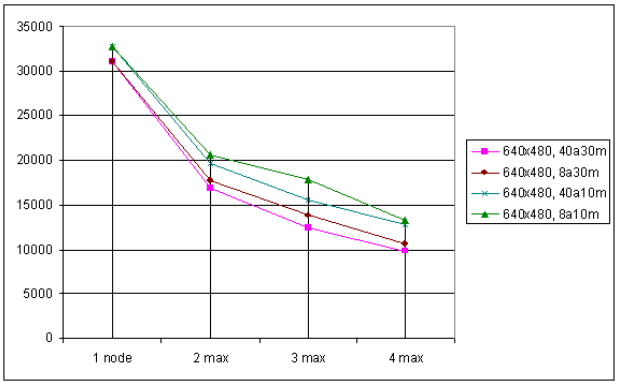

Figure 2, time by number of nodes

4.2 Size of Picture

As expected in theory doubling pixels requires twice the time. Therefore doubling one parameter of resolution results in quadruplication of time, which in matters of time can be mostly absorbed by quadruplication of nodes.

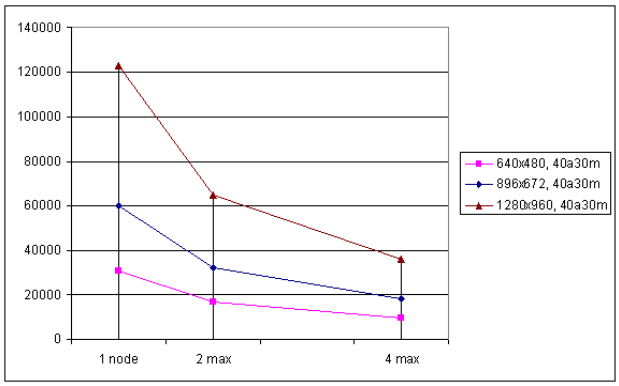

Figure 3, time by number of nodes

4.3 Number of Nodes

For low resolutions the time needed for calculation is not sufficient for distributing the work over more than four nodes, as each node should at least allocate three allocation-blocks. The theoretical maximum of scaling is printed as dotted line. Even in worst case the effective scaling factor of four nodes is greater than three, i.e. even three single nodes without DSM-overhead could not calculate each a third of this picture in the same time as four nodes with Plurix calculate the complete picture.

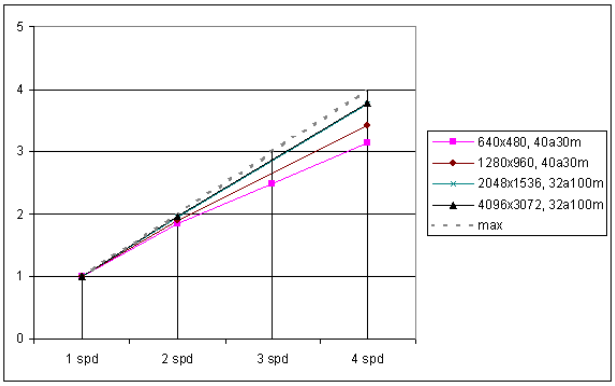

Figure 4, speed-up by number of nodes I.

Distributing a large picture exhibits linear scaling on more than four nodes, which was experimentally verified for up to twelve nodes.

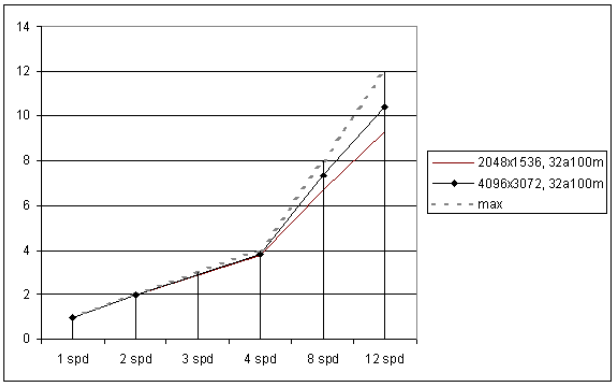

Figure 5, speed-up by number of nodes II.

5. Related Work

In 1985 L. Keedy presented an early idea of DSM [1]. In the following years a multitude of software and hardware level systems as well as hybrid architectures have been developed [12]. The proposed systems also differ in the used sharing unit: variables, objects, or pages. All these systems neither store all data nor code within the DSM. Typically, DSM data is allocated using special memory allocation functions and the programmer decides whether data is allocated in DSM or locally.

We are aware of other Java OSs like JX, JavaOS, and JOS. All these operating systems are not using a DSM in any way but are relying on traditional message passing for network communication.

The closest to our approach is the Kerrighed project adapting a Linux kernel for DSM operation including checkpointing and recovery [13]. Kerrighed uses sequential consistency which is also a strong model but code is not stored within the DSM and again only dedicated data objects. Furthermore, the research goals of Plurix and Kerrighed are different. The latter is mainly designed for parallel application running both DSM- and MPI-based algorithms whereas Plurix wants to explore new distributed applications for DSM systems.

Finally, Plurix is the first native operating system tailor made for DSM operation and transactional consistency was never used in any other existing DSM environment.

6. Conclusion and Future Work

In this paper we have evaluated the Plurix OS for the first time running itself within Distributed Shared Memory with a real application. The implementation is a proof of concept that it is possible to run OS code within a DSM and that even a strong consistency model can offer good scalability.

The measurements demonstrate potential high performance and scalability of transactional consistency [10] as it is implemented in Plurix [14]. Furthermore it simplifies programming in an environment of shared memory, as synchronization is implicit and need not to be considered by the programmer. There is no need for locks, barriers or other explicit calls to the runtime environment, because access to shared memory is automatically synchronized by the DSM protocol, which implements transactional consistency.

Additionally Plurix offers simplicity and safety of object orientated programming in familiar manner and an easy-to-use as well as type-safe kernel-interface [6].

Substantial work is pending on false sharing, fairness and lazy objects like frames of live video, which do not require strong consistency such as transactional consistency.

7. References

[1] J. L. Keedy and D. A. Abramson: “Implementing a Large Virtual Memory in a Distributed Computing System”. In Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, 1985.

[2] K. Li.: “IVY: A Shared Virutal Memory System for Parallel Computing”. International Conference on Parallel Processing, 1988.

[3] D. Mosberger: “Memory Consistency Models”. TR 93/11, Department of Computer Science, University of Arizona, Tucson, 1993.

[4] N. Wirth, J. Gutknecht: “Porject Oberon”. ACM Press, Addison-Wesley, New-York, 1992.

[5] M. Schoettner, “Persistente Typen und Laufzeitstrukturen in einem Betriebssystem mit verteiltem virtuellen Speicher”, PhD thesis, Ulm Univeristy, Germany, 2002.

[6] R. Goeckelmann, M. Schoettner, S. Frenz, P. Schulthess: “A Kernel Running in a DSM – Design Aspects of a Distributed Operating System”. Department of Distributed Systems, University of Ulm, 2003

[7] T. Bindhammer, R. Goeckelmann, O. Marquardt, M. Schoettner, M. Wende, P. Schulthess: “Device Driver Programming in a Transactional DSM Operating System”. In Proceedings of the Asia-Pacific Computer Systems Architecture Conference, Melbourne, Australia, 2002.

[8] P. Dadam: “Verteilte Datenbanken und Client/Server-Systeme”. Springer-Verlag Heidelberg, 1996.

[9] M. Schoettner, S. Frenz, R. Goeckelmann, P. Schulthess, “Checkpointing and Recovery in a transaction-based DSM Operating System”, to appear in: Proceedings of the IASTED International Conference on Parallel and Distribted Computing and Networks, Innsbruck, Austria, 2004.

[10] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, P. Schulthess: ”Optimistic Synchronization and Transactional Consistency”. In Proceedings of the 4th International Workshop on Software Distributed Shared Memory, Berlin, 2002.

[11] Tomas Lozano-Perez and Jovan Popovic: “Project 5: Ray Tracing”. http://graphics.lcs.mit.edu /classes/6.837/F01/Project05/project5.html, MIT, 2001.

[12] “A Comprehensive Bibliography of Distributed Shared Memory”. Technical Report TR96-17, Department of Computing Science, University of Alberta, 1996.

[13] http://www.kerrighed.org

[14] Homepage of Plurix: www.plurix.de