A Kernel Running in a DSM -

Design Aspects of a Distributed Operating System

R. Goeckelmann, M. Schoettner, S. Frenz and P. Schulthess

Department of Distributed Systems, University of Ulm, 89075 Ulm, Germany

goeckelmann@vs.informatik.uni-ulm.de

Abstract: The Plurix project implements an object-oriented Operating System (OS) for PC clusters. Communication is achieved via shared objects in a Distributed Shared Memory (DSM) - using restartable transactions and an optimistic synchronization scheme to guarantee memory consistency. We contend that coupling object orientation with the DSM property allows a type-consistent system bootstrapping, quick system startup and simplified development of distributed applications. It also facilitates checkpointing of the system state. The OS (including kernel and drivers) is written in Java using our proprietary Plurix Java Compiler (PJC) translating Java source code directly into Intel machine instructions. PJC is an integral part of the language-based OS and tailor-made for compiling in our persistent DSM environment. In this paper we briefly illustrate the architecture of our OS kernel which runs entirely in the DSM and the resulting opportunities for checkpointing and communication between applications and OS. We present issues of memory management related to the DSM-kernel and to strategies to avoid false-sharing .

Keywords: Cluster Operating System, Distributed Shared Memory, Reliability,
Object-Orientation, Single System Image

1 Introduction

Typical cluster systems are built on top of traditional operating systems (OS) as Linux or Microsoft Windows and data is exchanged using message passing (e.g. MPI) or remote invocation (e.g. RPC,RMI) strategies. As each node in a cluster is running its own OS with different configurations, the migration of processes is difficult, as it is unknown which libraries and resources will be available on the next node. Additionally, if a process is migrated to another node, the entire context including relevant parts of the kernel state must be saved and transferred. Because these OSs are not designed for cluster operation it is difficult to migrate kernel contexts [Smile] and as a consequence cluster systems typically redirect calls of migrated processes back to the home node, e.g. Mosix [Mosix].

Plurix is an OS specifically tailored for cluster operation and avoids these difficulties. The Distributed Shared Memory (DSM) offers an elegant solution for distributing and sharing data among loosely coupled nodes [Keedy],[Li]. Applications running on top of the Plurix DSM are unaware of the physical location of objects. A reference can either point to a local or to a remote memory block. During program execution the OS detects a remote memory access and automatically fetches the desired memory block. Plurix extends the DSM to a distributed heap storage, providing the benefit, that not only data but also the code segments of the programs are available on each node as they are shared in the DSM.

One of our major research goals is to simplify the development of distributed applications. Typically, DSM systems use weak consistency models to guarantee the integrity of shared data. This makes the development of applications hard, as each programmer must explicitly manage the consistency of the data by using the offered synchronization mechanism [TreadMarks]. Plurix uses a strong consistency model, called *transactional consistency* [Wende02] relieving the programmer from explicit consistency management.

Single-System-Image (SSI) computing architectures have been the mainstay of high performance computing for many years. In a system implementing the SSI concept, each user gains a global and uniform view on available resources and programs and provides the same libraries and services on each node in the cluster, which is very important for load balancing and migration of processes. We extend the

SSI concept by storing OS, kernel, and all drivers in the DSM. As a consequence we can implement a type-safe kernel interface and at the same time simplify checkpointing and recovery.

In 1990 Fuchs introduced checkpointing and recovery for DSM systems [Fuchs90]. Numerous subsequent papers discuss the adaptation of checkpointing strategies designed for message-passing systems ranging from global coordinated solutions to independent checkpointing with and without logging [Morin97]. However, the more sophisticated solutions have not been evaluated in real implementations because checkpointing is difficult to achieve in PC-clusters even under global coordination. If a checkpoint needs to be saved it is not sufficient to save the DSM context but also the local kernel context needs to be saved - which is not trivial. Plurix avoids these drawbacks by storing OS and applications in the DSM.

The remainder of the paper is organized as follows. The design of Plurix is briefly presented in section 2. We then describe the advantages of a type-safe kernel interface. In the sequel we describe the benefits of running the kernel within the DSM. Extending the SSI provides additional advantages for the checkpointing, which are described in section 5. Finally, we present measurements and give an outlook on future work.

2 Design of Plurix

Plurix implements SSI properties at the operating system level, using a page-based distributed shared memory. According to the SSI concept all programs and libraries must be available on all nodes in the cluster. Therefore Plurix uses a global address space shared by all nodes and organized as distributed heap storage (DHS) containing both data and code. To share the programs in the DHS reduces redundancy concerning code segments and makes the administration of the system easier.

2.1 Java-based Kernel and Operating System

Plurix is entirely written in Java and works in a fully object oriented fashion. The development of an operating system requires access to device registers which is not possible in standard Java. For this reason we have developed our own Plurix Java Compiler (PJC) with language extensions to support hardware

level programming. The compiler directly generates Intel machine instructions and initializes runtime structures and code segments in the heap. Traditional object-, symbol-, library- and exe-files are avoided. Each new program is compiled directly into the DHS and is thereby immediately available at each node.

Plurix is designed as a lean and high speed OS and therefore able to start quickly. The start time of the primary node, which creates a new heap (installation of Plurix) or restarts an preexisting heap from the PageServer, is less than one second. Additional nodes, which only have to join the existing heap, can be started in approximately 250 ms. This quick boot function of Plurix is helpful to guarantee fast node and cluster start-up time, which helps to avoid long downtimes in case of critical errors.

2.2 Distributed Shared Memory

The transfer of the DHS-objects from one cluster node to a new one is managed within the page-based distributed shared memory (DSM), and takes advantage of the Memory Management Unit (MMU) hardware. The MMU detects page faults, which are raised if a node requests an object on a page which is not locally present. Each page fault results in a separate network packet which contains the address of the missing page (PageRequest). This packet is broadcast to all nodes in the cluster (Fast Ethernet LAN) and only the current owner of the page send it to the requesting node.

An important topic in distributed systems is the consistency of shared and replicated objects. In Plurix this is synonymous to the consistency of the DSM. Plurix offers a new consistency model, called transactional consistency, which is described in the following section.

2.3 Consistency and Restartability

Unlike traditional systems, Plurix does not burden the programmer with the consistency of the DHS-objects. All actions in Plurix are encapsulated in transactions. At the start of a transaction, write access to pages are prohibited. If a page is written, the system creates a shadow image of it and then enables write access. Additionally, the system logs the pages for which shadow images were created. At the end of a transaction (commit phase) the addresses of all modified pages are broadcasted and all partner nodes in the

cluster will invalidate these pages. If there is a collision with a running transaction on another node, it is aborted and eventually restarted.

In case of an abort all modified pages are discarded. Since there is a shadow image for each modified page, the system can reconstruct the state of the node at the time just before the transaction has been started. A token mechanism is used to ensure, that only one node is in the commit phase at a time. The token is passed using a first wins strategy. To improve fairness further commit strategies will be developed.

2.4 False Sharing and Backchain

All page-based DSM systems suffer from the notorious false-sharing syndrome. False-sharing occurs, if two or more nodes access separate objects which nevertheless reside on the same page. If a node writes to such an object, all other nodes are forced to abort their current transaction and restart it later. As these objects are not shared, such an abort is semantically unjustified and unnecessarily slows down the entire cluster. To handle this problem relocation of DSM-objects from one physical page to an other is required. When an object is relocated, all pointers to this object are adjusted. Due to the substantial network latency in the cluster environment, it is not possible to inspect each object whether contains a pointer to the relocated object. To adjust the affected references, Plurix uses the *backchain* [Traub]. This concept links together all references to an object, by recording the addresses of these pointers (see fig. 1). All references to a relocated DSM-object are found in the backchain. To reduce invalidations of remote objects when a new backchain entry is inserted, references on the stack are not tracked in the backchain.

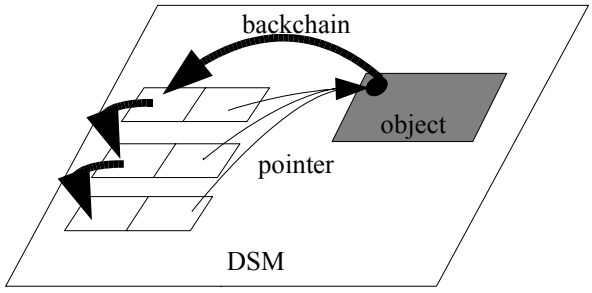


Figure 1 The Backchain Concept

2.5 Garbage Collection

The previously described backchain concept can also be used to simplify a distributed garbage collection (GC). A Mark-and-Sweep algorithm should not be used in a DHS-environment, because it is either very difficult to implement (incremental Mark-and-Sweep) or it would stop the entire cluster while collecting garbage. Copying GC algorithms will unduly reduce the available address space - only reference counting algorithms appear feasible. The backchain can be used as a reference counter. If an object contains an empty backchain, no references to this object remain. This is equivalent to a reference counter of 0, so in this case the object is garbage and can be deleted. Because stack references are not included in the backchain, the GC may only run if the stack is empty. Between two transactions this condition is always true, and that the GC task can be run as a regular Plurix transaction.

3 A Type-Safe Interface for a DSM-Kernel

The SSI concept requires, that all nodes in the cluster have the same programs installed. In a distributed environment the easiest why to achieve this goal is to share not only data but also the code of the programs, for this reason the Plurix extends the DSM to the DHS. In this case it is mandatory to protect the code segments from unwanted modification either by corrupted pointers or by malicious attacks. This can be achieved by using a type-safe language like Java. Language-based OS development has been successfully demonstrated by the Oberon system [Wirth]. The requirement for type safety in the DSM affects also the interface to the OS. As data in the DSM is represented by objects and these data must be passed to the kernel, either the objects must be serialized before they are used as parameters or the kernel must be able to accept objects.

3.1 Traditional Kernel interfaces

Traditionally, distributed systems are implemented as a middleware layer on top of a local OS, such as Linux or Mach, which are mostly written in C and therefore do not provide objects. The communication between the distributed system and the local OS takes place using primitive data types or structures. If the kernel cannot handle objects as such, they are serialized (and data items are copied) before being passed to

the kernel. This kind of raw communication does not provide type checks for parameters and signatures by the runtime environment. Hence no type-safe calls of kernel methods are possible and each kernel method has to check explicitly its parameters to avoid runtime errors.

3.2 Benefits of a Type-Safe Kernel Interface

To reduce programming complexity and to increase system performance we recommend to pass typed objects to the kernel. This was part of the motivation to create Plurix as a stand alone OS not as a middleware layer. Since the kernel of Plurix is written in Java and easily handles objects, a type-safe communication between the DSM applications and the OS is natural. All Java types and objects, can be handed to the kernel methods. The programmer has no need to pay attention to the type of the passed object because this is checked by the compiler and in some cases by the runtime environment. Further on there is no reason to serialize objects which are used as parameters for kernel methods, so the performance of the entire system increases.

Another benefit of using objects as parameters is that the object respectively the data included in these objects need not be copied. The kernel method obtains a reference and accesses the object directly. This increases the system performance again.

3.3 Inter Address Space Pointers

In traditional systems there are at least two different address spaces, one for the kernel and at least one for user applications. As the kernel methods are always needed on each node the straight-forward way of implementing the system would be to place the kernel in the local address space. These local addresses are not shared with other nodes, and each node in the cluster can use them in different ways. In this case a separation between the kernel and user address space would mean to differentiate between the local- or NonDSM- and the DSM-address-space. If in such an environment objects are used as parameters, some references will point from the Non-DSM into the DSM address space. References which points from the Non-DSM into the DSM reduces the performances of the cluster, as they inhibit the relocation of objects so that avoidance of false-sharing and memory fragmentation is prevented. The reason being that the

backchain entries are not longer unambiguous when an object migrates to another node and is then relocated from one DSM address to another. If an object is referenced by a Non-DSM-object, the Backchain leads into the local memory of the node. As addresses in the local memory are not unique, the pointer can not be adjusted, as it is not possible to detect which local memory area is specified by this backchain entry. The correct reference to this object can not be found and an adjustment of the memory location which is specified by the backchain will lead to invalid pointers or even destroyed code segments (see figure 2).

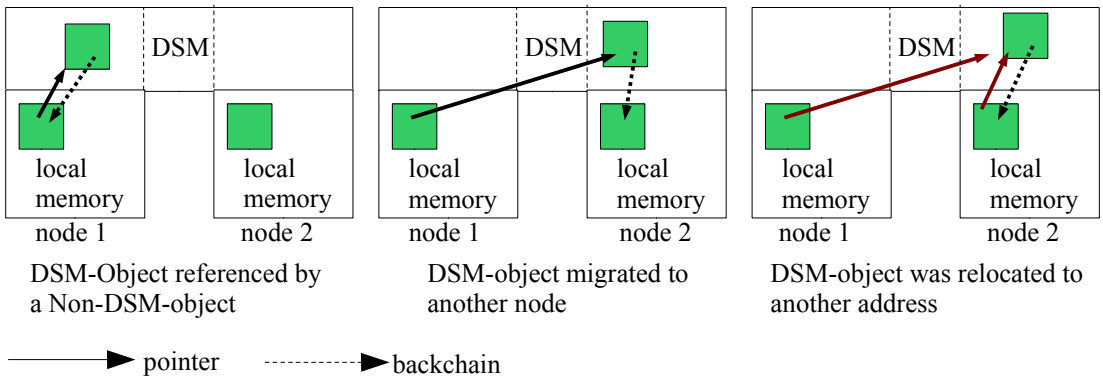


Figure 2 Migration and subsequently relocation of a DSM-object

As long as DSM-objects are relocatable, references from the Non-DSM into the DSM address space are not possible, as they could lead to dangling pointers or destroyed code segments. To solve this problem it would be possible to prevent relocation of DSM-Objects, which are referenced by Non-DSM-object. As it is not possible to specify, which objects are used as parameters for kernel methods, nearly all objects in the DSM could not be relocated and so the performance of the cluster will be impaired because false sharing and fragmentation of the memory can not be handled. Therefore direct pointers from the Non-DSM into the DSM-address-space must be avoided.

Another interesting question is how kernel methods can be called from DSM applications. Two alternative methods are conceivable:

1. Software Interrupt: Like in most traditional systems, kernel methods may be called using kernel- or system-calls. These are software interrupts which request a specific function from the kernel. If kernel-calls are used to communicate between the DSM applications and the operating system there are no "address space spanning" pointers but the question arises how to pass parameters from the DSM to the kernel, as the software interrupt itself cannot accept parameters. One possible solution is to pass data to the kernel through a fixed address. If an object is used as a parameter, this address would contain the pointer to the object which should be used. As each kernel method requires different parameters, this object must be of a generic type and thereby each object can be passed. Each kernel method has to check the given object if it is type compatible with the expected one as this could not be handled by the runtime environment. This rises the complexity for system programmers and makes the system vulnerable to faults by simultaneously reducing the performance and the possibilities of parameter passing.
2. Object oriented invocation: Kernel methods are invoked in an object oriented fashion via direct pointers to the requested kernel class. This implies that all kernel classes and their methods have to reside at the same addresses on each node in the cluster. This is necessary as each application can only have one pointer to a kernel class. Should they reside at different addresses, these references would point to invalid addresses and the corresponding kernel methods could not be called correctly on some nodes (see figure 3). If direct pointers are used each node in the cluster must run the same kernel, and such a kernel can never be changed during runtime. Consequently all pointers in the applications, which reference kernel methods require adjustment. To achieve this, all kernel methods must contain a backchain which points from Non-DSM into the DSM and thereby the above described problems will occur.

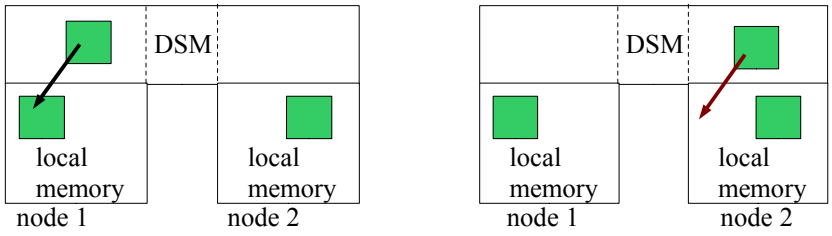


Figure 3 Invalid reference to kernel methods

Both techniques give rise to an additional problem. The compiler is running in the DSM and any new program is automatically created in the DSM. If the new program is a device driver (which typically resides in kernel space) the code segments must be transfered from the DSM into the Non-DSM address space and this must occur simultaneously on each node.

Our implemented solution which solves all the challenges above is to remove the kernel from the local memory address space and move it into the DSM. Further benefits of this approach are described in the following section.

4 Extending the Single System Image Concept

We elaborate the SSI concept by moving the OS and the Kernel into the DSM. The local memory is only used for a few state variables for the network device drivers and -protocol and for the so called Smart-Buffers, which help to bridge the gap between not restartable interrupts and transactions [Bindhammer].

4.1 Benefits of a kernel running in the DSM

If the kernel runs in the DSM parameter passing between applications and kernel is elegant and all objects can be used as parameters. Kernel methods are called directly as described in section 3.3. There are no references pointing from one address space to the other. Since all device drivers now reside in the DSM even the problem of transferring newly compiled drivers from the DSM into the kernel space vanishes. Because the code segments of the kernel methods are in the DSM redundancy is avoided. Further benefits from this concept, especially for system checkpointing are described in Section 5.

Some interesting questions surfaced when moving the kernel into the DSM, but before we describe these topics and our corresponding solutions we describe the memory management of Plurix and the allocation mechanism for new objects as this is important for our solution.

4.2 Distributed Heap Management

A basic design topic of the Plurix system is the page-based DSM, raising the false sharing problem. The allocation strategy of the memory management must try to avoid false sharing wherever possible. Furthermore collisions during the allocation of objects in the DHS must be avoided, as such a collision will abort other transactions and thereby serialize all allocations in the cluster. To achieve those goals, Plurix uses a two stage allocation concept consisting of *allocator*-objects and a central memory manager. The latter is needed, as the memory has to be portioned to the different nodes in the cluster. This division must not be static, as this would reduce the maximum size of the objects.

The memory manager is used to create allocators and large objects. As the allocator must have at least the same size as the new object which should be created, the usage of allocators for large objects would lead to large allocators and thereby to a static fragmentation of the heap. The alternative for this is to limit the size of the allocators and thereby the maximum size of the DHS-objects which is unacceptable.

Allocator-objects represent a portion of empty memory. The size of an allocator is reduced for each allocated object. If it is exhausted, the Allocator is discarded and a new one is requested from the central memory manager.

4.2.1 Allocation of Objects

If a new object is requested, the memory management first decides if the object is created by the corresponding allocator or by the memory manager. This decision depends on the size of the object. Each object which is greater than 4KB is directly allocated by the memory manager. To avoid false sharing on these objects, their size is increased to a multiple of 4 KB (page granularity of the 32-bit Intel architecture). As all objects which are allocated by the memory manager have a size of a multiple of 4 KB, each object starts at a page border and consumes N pages. Therefore these objects do not co- reside with other objects on the same page.

Objects which are less than 4 KB are created by an allocator. As each node has its own allocator, collisions can only occur if a large object is allocated or if an allocator is exhausted and a new one must be created. The measurements in section 6 show, that the size of most of the objects in Plurix are less than

4KB, so large objects are rarely allocated. The collisions which occur during these allocations are tolerable most of the time.

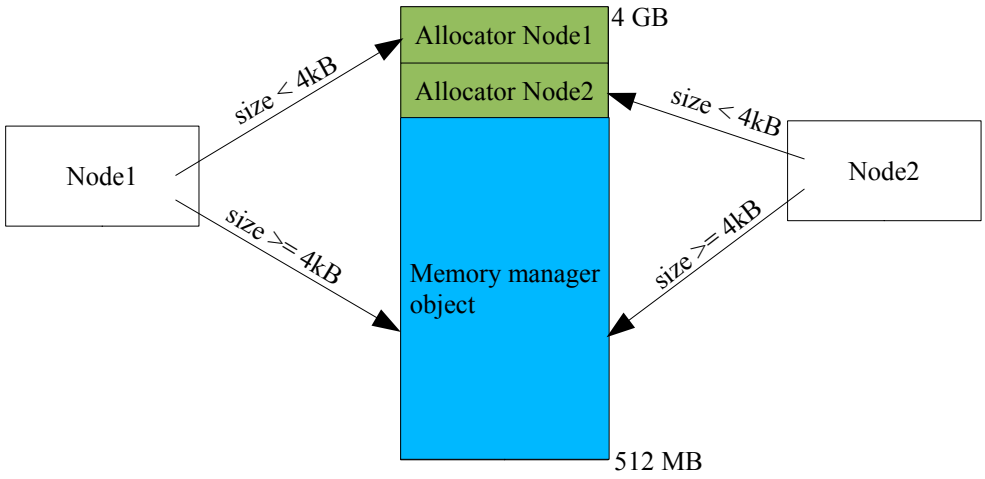


Figure 4 Allocation of objects

The benefit of the two level allocation of objects is that small objects from one node are clustered in the memory. As a consequence collisions do not occur during the allocation of small objects and are rare if large objects are allocated. As large objects are not allocated within the allocator, its size can be limited, without limiting the maximum size of the objects. No static division of the memory is needed and therefore no static fragmentation is created.

4.2.2 Reduction of False Sharing

Generally speaking objects can be divided into two categories: Read-Only (RO) and Read-Write (RW) objects. False-sharing on RW-objects is reduced by the mechanism described above. To further reduce false-sharing it is reasonable to make sure, that RO-objects like code segments and class descriptors without static variables do not co-reside with RW-objects on the same page, as this would lead to unnecessary invalidations of the RO-objects due to false-sharing. Code segments are only written during compilation by the compiler. If these objects would be indiscriminantly allocated, they could reside on the same pages as the RW-objects of the node which is currently running the compiler. To avoid this, Plurix provides additional allocators for RO-objects.

4.3 Protection of SysObjects

If the entire system is running in the DSM, some code segments and instances of classes must be protected against invalidation, as these objects are vital for the system. The objects which must always be present on a node are called SysObjects. These are nota bene all classes and instances concerning the Page-Fault-handler, the DSM protocol and the network device drivers. As these objects reside in the DSM they might be affected by the transaction mechanism and in case of a collision on such a page, the page would be discarded and the system will hang, as the node is no longer able to request missing pages.

The protection of SysObjects against invalidation is easy to achieve just by defining two additional allocators. SysObjects are either code segments or instances of SysClasses. As described above, code segments are only written during compilation otherwise they are read only. Additionally, code segments should not co-reside on the same pages as RW-objects as this would lead to false-sharing and therefore a special allocator is used. The compiler will create the new kernel classes in a different memory area. Afterward update messages need to be sent to all nodes in the cluster, to replace old classes and instances by new ones. To achieve this, it is sufficient to make sure that such an allocator is only used by the current compilation and after that the remaining part of the last used page is consumed by a Dummy-SysObject.

RW-SysObjects are instances of SysClasses which are meaningless for all nodes except that one, that has created the instance. For this reason RW-SysObjects are not published through the global name service. Therefore no other nodes can access a RW-SysObject. The only case where a RW-SysObject could be invalidated is as a result of false-sharing. To prevent this, each node acquires a SysRW-Allocator during the boot phase. All instances of SysClasses are allocated in this private allocator, so that there are only SysObjects from one node on the same page.

These two additional allocators and the described techniques to use them are sufficient to protect all SysObjects against invalidation at run time.

4.4 Local memory for State Variables

State variables of the DSM protocol and the network device drivers must outlast the abort mechanism, as these variables are needed to handle the abort mechanism. If they would be reset the current state of the protocol and the network adapter would be lost. The network device driver would never be able to receive the next packet as the receive-buffer pointer would also be reset. Also the protocol contains a sequence number for the messages to make sure, that no vitally important message is lost. If the state variables are reset, the protocol will receive messages from the future. In this case it would not be possible to decide if this number is invalid due to an abort or if the node has missed important network packets.

As the protocol is not a device driver, its current state variables can not be read from the hardware registers, as it is possible for normal (not network) device drivers. Hence these variables must be stored outside the DSM address space. For device drivers and the protocol, the kernel provides special memory areas in the local memory in which state variables are stored. To access these areas, Plurix provides “structs“, allowing to address raw memory much like the variables in an object. “structs“ are also used to access the memory mapped registers of devices. As Structs may not contain pointers and are not referenced by pointers no problems with address space spanning pointers arise.

4.5 Restart of device drivers

In case of an abort the state of the entire node is reset to the state just before the current transaction was started. Devices can not be automatically reset and the device driver programmer must implement an *Undo*-method, which is called by the system in case of an abort. This method has to ensure that both the state of the hardware and that of the state variables in the device driver object are reset. To make this possible the state of all devices before the transaction must be conserved.

An example for such an Undo-method is shown for graphics controller devices. In this case the current On- and Off-screen memory-area on the display adapter must be reset. Since between two transactions the On-and Off-screen contains the same data, it is sufficient to reset the Off-screen memory and afterward copy this value to the On-screen area. This is easy to implement as most graphics controllers contain substantial amounts of memory for textures and vertices. A small part of this memory can be used to save

the committed state of the graphic controller. After the commit phase, the current On-Screen area is copied into this separate memory area and can be restored if necessary.

The serial-line controller is more difficult to handle. This controller sends all data if it receive it from the system. In case of an abort it is not possible to “undo” the sent data. For this problem there are two possible solutions. Either the affected application is able to handle duplicated data or the driver has to use smartbuffers. Data in this special buffer type are invisible to the device until the commit phase, so the device can only access committed and therewith value data.

5 Checkpointing and Recovery

State of the art PC-Clusters are built using Linux or Microsoft Windows but implementing checkpointing and recovery in these operating systems is difficult because it is not sufficient to save the process context but also the local kernel context needs to be observed. The latter includes internal data structures, open files, used sockets, pending page requests, ... which can be read only at kernel level. Resetting the kernel and process context in case of a rollback is also challenging because of the complex OS architectures. As a consequence taking a checkpoint is time consuming and checkpointing intervals are quite large, e.g. 15-120 min. for the IBM LoadLeveler.

By extending the Single System Image concept we avoid these drawbacks. Storing the kernel and its contexts in the DSM makes it easy to save this data. Rollback in case of an error is no problem in Plurix because the OS and all applications are designed to be restartable anyway.

5.1 Current Implementation

A central fault-tolerant PageServer stores consistent heap images in an incremental fashion on disk. Between two checkpoints the PageServer uses a bus snooping protocol to intercept transmitted and invalidated memory pages to reduce the amount of data to be retrieved from the cluster at the next checkpoint. If a checkpoint must be saved the cluster is stopped and the PageServer collects invalidated pages that have not been transmitted since the last checkpoint. All memory pages are written to disk

synchronously. We have implemented a highly optimized disk driver that is able to write about 45 MB/s. An early performance evaluation of our PageServer can be found in section 6.

Because the kernel and its context reside in the DSM we must not save node local data. Furthermore, we have no long running processes or threads with preemptive multitasking that need to be checkpointed. Currently, we use a cooperative multitasking scheme for executing short transactions. A transaction is executed by a command or periodically called from the scheduler. Long running computations have to divided in sub transactions manually. In case of an error the node can perform a reboot and fetch required memory pages again from the DSM from the last checkpoint.

5.2 Fault-Tolerance

We support clusters running within a single Fast Ethernet LAN and assume a fail-stop behavior of nodes. Most DSM systems typically use a reliable multicast or broadcast facility to avoid inconsistencies caused by lost network packets. Because of the low error probability of a LAN we are not willing to impose the overhead by a reliable communication during normal operation. Instead we rely on a fast error detection, fast recovery, and the quick boot option of our cluster OS.

As described in 2.3 our DSM implements transactional consistency and committing transactions are serialized using a token. We introduce a logical global time (a 64-Bit value) incremented each time a transaction commits. In the latter case the new time is broadcasted to the cluster and each node updates its time variable. A node can immediately detect if it missed a commit and ask for recovery. If the commit message cannot be sent with one Ethernet frame, the commit number is incremented for each commit packet. Thus we avoid inconsistencies if a node did miss a packet of a multiple packet commit. Furthermore, any page or token requests always includes the global time value of the requesting node. If such a request contains an out-of-date commit number it is not processed but recovery is started. Thus a node that missed a commit is note able to commit a transaction because it is not granted the token.

If a single node fails temporarily it can reboot and join the DSM again. If the PageServer detects missing pages during the next checkpoint that were lost because of the a node failure the cluster is reset to

the last checkpoint. If a multiple nodes fail temporarily or permanently the same error detection scheme works, too.

The network might be partitioned temporarily into two or more segments. Only one token and one PageServer is available in any of these segments. Nodes within the segments send page and token requests. If either request cannot be satisfied the segment tries to recover by contacting the PageServer. Only the segment with the PageServer can recover the others have to wait until the PageServer becomes available again.

We plan to implement a distributed version of our PageServer two avoid a bottleneck and to replicate data stored on the PageServers to tolerate failures of PageServers, too. We also plan to introduce a asynchronous checkpointing scheme to avoid stopping the cluster during checkpointing operation. Dependency tracking will also be investigated to restart only affected nodes in case of a failure.

6 Measurements

The performance evaluation is carried out on three PCs interconnected by a Fast Ethernet Hub. Each node is equipped with a RLT8139 network card and an ATI Radeon graphic adapter. Only the first machine (with Celeron-CPU) is featured with a hard disk (IBM 120GB, max disk write throughput without network 45 MB/s) and acts as PageServer.

Table 1. Node configuration

Node	***CPU***	***RAM***
1	Celeron 1.8 GHz	256 MB DDR RAM at 266 MHz
2	Athlon XP2.2+ at 1.8 GHz	256 MB DDR RAM at 333 MHz
3	Athlon XP2.0+ at 1.66 GHz	256 MB DDR RAM at 333 MHz

6.1 General System Measurements

We have tested the startup time of the above described cluster nodes. The results are split into the time which the kernel needs and the time which is needed to detect and start the hardware such as HD, mouse and keyboard. The nodes have been started with and without harddisc and the time difference is about 540 ms during which we have to wait for the harddisc to answer.

Table 2. Startup times (in ms)

Node	Startup as Master	Kernel time	Startup as Slave	Kernel time
1	791	254	240	234
2	780	248	238	233
3	792	254	239	234

The kernel allocates 2787 objects if running as master and 518 objects if running as slave. It takes approximately 3 microseconds to allocate an object and additional 0.5 microseconds to assign a pointer to an object. To get the kernel from the DHS a slave node must request 284.

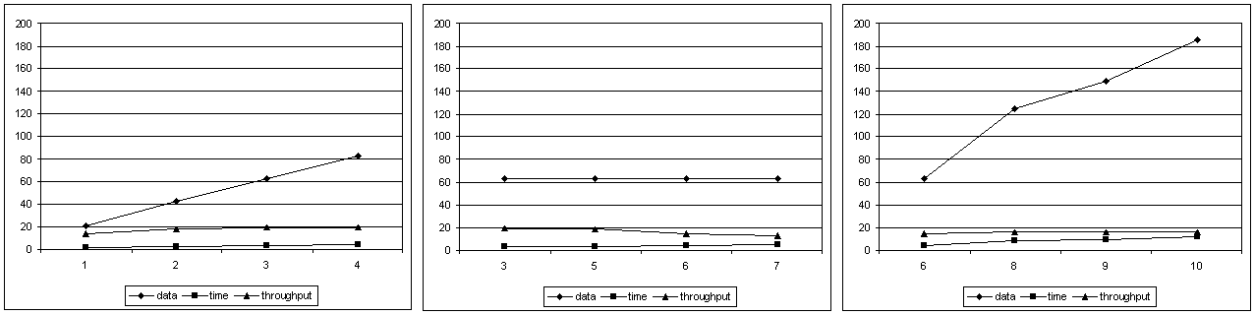
To show the correlation of changed heap size, heap spreading and time to save a checkpoint, ten measurements were made. Comparison of several measurements is needed for predications about speed of hard disk, performance of implemented software and latency caused by network. In the following table for each measurement the configuration (single station or cluster) and heap spreading is given. The PageServer creates consistent images of the complete heap containing both user data (node1 – node3) and operating system. The latter is contained in “saved data”.

Table 2. Measurements

#	*nodes*	*Node 1*	*Node 2*	*Node 3*	*Saved data*	*Time to save to disc*	*Throughput (resulting disc write bandwidth)*
1	1	20 MB	-	-	21,4 MB	1639 ms	13,7 MB/s
2	1	40 MB	-	-	42,5 MB	2491 ms	17,5 MB/s
3	1	60 MB	-	-	63,0 MB	3371 ms	19,1 MB/s
4	1	80 MB	-	-	83,4 MB	4321 ms	19,7 MB/s
5	1, 2, 3	60 MB	0 MB	0 MB	63,1 MB	3422 ms	18,9 MB/s
6	1, 2, 3	20 MB	20 MB	20 MB	63,1 MB	4476 ms	14,4 MB/s
7	1, 2, 3	0 MB	28 MB	32 MB	63,1 MB	4971 ms	13,0 MB/s
8	1, 2, 3	40 MB	40 MB	40 MB	124,6 MB	8049 ms	15,8 MB/s
9	1, 2, 3	48 MB	48 MB	48 MB	149,1 MB	9540 ms	16,0 MB/s
10	1, 2, 3	60 MB	60 MB	60 MB	186,0 MB	11707 ms	16,3 MB/s

In comparison of measurement 1-4 we see an increase of throughput in consequence of increased data size. Measurements 3, 5-7 have same size of saved data, so decreased throughput depends on network latency. Comparing measurements 6, 8-10 approves nearly constant

throughput. The slight improvement for increased data size is due to faster saving of local data. The following chart shows these three comparisons:



7 Experiences and Future Work

Moving the kernel into the DHS and therewith elaborating the SSI concept made it possible to create a type-safe kernel interface and to solve the problem of address space spanning pointers. Additionally, checkpointing is made much easier and the question in which way kernel methods should be called is answered.

The current version of Plurix is running stable in the cluster environment, without collisions during allocation. The usage of the allocator strategy inhibits false sharing if no applications which share objects are running. As soon as objects are created by an application and shared with other nodes, the allocation mechanism is not able to prevent false sharing but we are working on a monitoring tool to detect false sharing. Relocation of objects to dissolve false sharing is currently available.

Plurix uses a distributed garbage collection algorithm which is able to detect and collect garbage (including cyclic garbage) without stopping the cluster. The detection algorithm for cyclic garbage works error free but currently there is no information which object could be cyclic garbage so each object in the DHS must be checked.

The consistency of the DHS is ensured by the PageServer, which uses linear segment technique to save all changed pages. This includes data and code objects of user applications as much as the OS. In the current implementation the speed of saving the complete heap is limited by network throughput and not by OS or hard disc. For this reason it is necessary to save the state of the cluster continuously which could be achieved by some minor changes, regarding the mechanism of detecting missing pages.

8 References

[Mosix] Barak A. and La'adan O., The MOSIX Multicomputer Operating System for High Performance Cluster Computing , Journal of Future Generation Computer Systems, Vol. 13, No. 4-5, pp. 361-372, March 1998.

[Wirth] N. Wirt and J. Gutknecht, „Project Oberon“, Addison-Wesley, 1992.

[Traub] S. Traub, “Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen”, PhD thesis, University of Ulm, 1996.

[TreadMarks] Amza C., Cox A.L., Drwarkadas S. and Keleher P., „TreadMarks: Shared Memory Computing on Networks of Workstations“, Proceedings of the Winter 94 Usenix Conference, pp. 115-131, January 1994.

[Fuchs90] Kun-Lung Wu and W. Kent Fuchs, „Recoverable Distributed Shared Virtual Memory”, IEEE Transactions on Computers, 39(4):460-469, April 1990.

[Morin97] C. Morin, I. Puaut, “A Survey of Recoverable Distributed Shared Virtual Memory Systems”, IEEE Transactions on Parallel and Distributed Systems, Vol. 8, No. 9, September 1997.

[Keedy] J.L. Keedy, and D. A. Abramson, “Implementing a large virtual memory in a Distributed Computing System”, in: Proc. of the 18th Annual Hawaii International Conference on System Sciences, 1985.

[Li] K. Li, “IVY: A Shared Virtual Memory System for Parallel Computing”, In Proceedings of the International Conference on Parallel Processing, 1988.

[Wende02] 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, Germany, 2002

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

[Simle] http://os.inf.tu-dresden.de/SMiLE/