

Transactional Distributed Memory Management for Cluster Operating Systems

N. Kaemmer and S. Gerhold and T. Baeuerle and P. Schulthess

Institute of Distributed Systems

Ulm University

James-Frank-Ring 027, 89069 Ulm, Germany

Phone: (49) 731-5024144 Fax: (49) 731-5024142 E-mail: nico.kaemmer@uni-ulm.de

Abstract - Memory management is a crucial component of distributed operating systems. Our paper presents design and functionality of the memory management package for Rainbow OS (a 64-Bit transactional distributed memory (TDM) operating system for PC-clusters). To support efficient parallel computing on distributed objects several consistency models and multicore operation is provided. Even when using relaxed consistency models for selected application data the global heap structure and all object references remain consistent and safe. Coupled to the different consistency models several alternatives for clusterwide garbage collection are offered.

I. INTRODUCTION

Rainbow OS [1] is a transactional distributed memory operating system for PC-clusters, designed for 64-Bit multicore architecture. Rainbow OS is completely implemented in Java and compiled into native code with a special compiler [4]. So Rainbow OS offers system and application programmers the advantages of a type safe language and yet providing fast runtime performance. For memory management reasons, the design of Rainbow OS can be separated into two parts one for the local memory management, integrated in the so-called local kernel, being responsible for the lowest address region of our 64-Bit address space and implements several basic system functions, too. The local kernel of Rainbow OS is local and exclusively usable for each node of the cluster. We describe the design and functionality of the locally memory management of the local kernel in the next chapter of this paper. In contrast to the local kernel the so-called distributed kernel of Rainbow OS is responsible for the rest of the higher 64-Bit address space and includes additional shared system functions. The distributed kernel is completely distributed and shared by all nodes of the cluster. One challenge is to guarantee the consistency of it and the shared memory management for correct functionality of the cluster and its nodes. Therefore Rainbow OS implements a transactional consistency mechanism to ensure these requirements. In addition to the Transactional Consistency Rainbow OS provides other consistency models. So application programmers can use their own consistencies for their applications, being suitable for their requirements. To protect Rainbow OS and its type system other consistency models are only usable for primitive data types (e.g. integers, longs and so on) until now. Providing several consistency models Rainbow OS uses a special object design, being different to objects of Sun Java. Further details are shown in the next chapters of this paper.

II. MEMORY MANAGEMENT

One challenge in the design of Rainbow OS is the distribution of all data in the cluster as far as possible. So Rainbow OS does not only share simple object structures but also code, which is usable by all nodes. Due to different hardware characteristics or system critical components not all data can be shared in the cluster. Facilities for the boot process or for network communication must be exclusively available at start for every node, before it is able to join the cluster, if it exists. Therefore system or any hardware specific components (e.g. network adapter) are implemented in the local kernel of Rainbow OS, which is bounded statically, providing best runtime performance. After the startup process the local kernel initializes all necessary drivers and a small subset of system tools before the node can join the cluster. Drivers and memory management of the local kernel are exclusive for every node and not shared in the cluster. So every node and its local kernel are comparable with a single station system.

In contrast to the local kernel the distributed kernel of Rainbow OS is completely distributed and therefore shared by all nodes. After the boot process and initializing of the local kernel the distributed kernel will be loaded from the shared memory of the cluster. If there is no cluster it can be loaded from a special sever (so-called boot server) or a separate storage medium (e.g. CD-ROM). The distribution of the distributed kernel implicates one shared memory management for all nodes in the cluster. So every node has the same view of the shared memory (single system image) and uses the same code and data structures of the distributed kernel. In the next two paragraphs we will give an overview about the memory management of local and distributed kernel. To guarantee a correct functionality of the cluster and especially of the distributed memory management the distributed kernel and its data are under control of Transactional Consistency (see paragraph II.D).

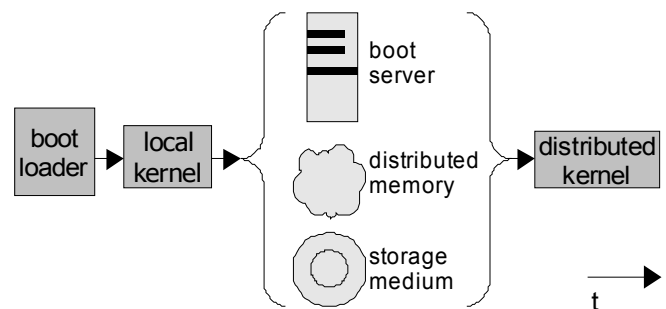


Figure 1: Startup process of Rainbow OS

A. Local Memory Management

The local memory management of Rainbow OS primarily maintains the physical pages of each node, their page tables and offers system or driver programmers two memory pools for memory allocation. Both memory pools are logically placed between the local kernel (see figure 2) and the distributed kernel (see paragraph II.C).

The interface for programmers to allocate memory is the well known `new()` of Java. In difference to Sun Java Rainbow OS uses another internal structure of its objects. Every object in Rainbow OS can be separated into two parts, one for primitive data types (so-called scalars) and one for references. References to objects separate these two regions, references are stored below and scalars above (see figure 3). A specific characteristic of Rainbow OS is the opportunity to store scalars at another location in memory, outside of the proper object. These so-called indirect scalars are not used in the local memory management, keeping it small and avoiding unnecessary complexity, offering best performance and a fast startup process. Further information on the structure and usage of indirect scalars are shown in paragraph II.C.

Another function of the local memory management is the creation and the administration of page tables for the logical address space of Rainbow OS. Using a 64-Bit architecture it is inappropriate and impossible to create and store all page tables for the entire 64-Bit address space with the current capacity of memory in our cluster nodes. Therefore the local memory management provides an on demand allocation and creation of new page tables, keeping the set of page tables small at startup. Furthermore parts of page tables can be removed to cleanup memory, if the corresponding address space is unused. Due to allocation and creation on demand, the memory management reserves at startup a specific area of logical address space to map and access all page tables, if necessary.

B. Multicore

Rainbow OS supports multicore processors with a 64-Bit architecture. Currently, only one core is able to join the cluster and works truly distributed, the others are only locally available at the moment. Our goal is to support each core of a multicore processor in this way. Consequently by each core of a processor in each cluster PC represents an independent node in the cluster. To enable multicore working in cluster some additional requirements have to be met. Some hardware components are unique in every cluster PC (e.g. physical memory or network adapter) and have to be synchronized between all cores of one processor, to guarantee correct processing. For example the physical memory management has to be synchronized or partitioned, to avoid duplicated allocation of physical

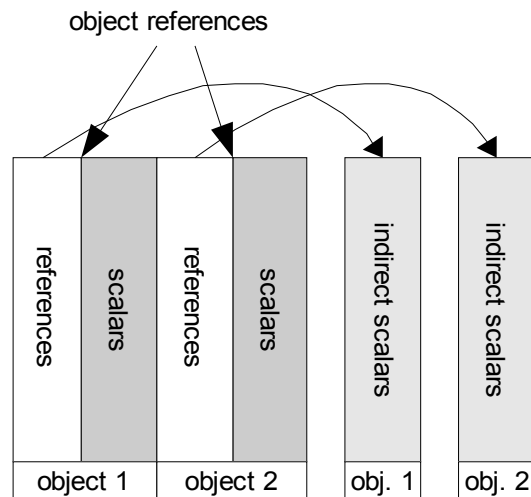


Figure 3: Object structure

pages. Furthermore each core could use another section of shared memory, so their page tables can differ and are not completely sharable by all cores, especially in regard to the mechanism of Transactional Consistency. Further research will focus on the complete support of multicore cpus mentioned before.

C. Distributed Memory Management

As mentioned in the previous paragraphs Rainbow OS has a local and a distributed memory management. The latter is implemented in the distributed kernel and completely distributed in the cluster. Due to the existence of two kernels, a local and a distributed, and the fact that not every driver or system component can be shared by all nodes, an additional communication facility between both kernels is necessary. Therefore Rainbow OS uses a so-called Integer-Interface with own data buffers to communicate between the local and the distributed kernel widening the scope in which code can be shared by nodes of the cluster. So for example drivers can be implemented in the distributed kernel of Rainbow OS, too. Furthermore Rainbow OS supports several consistency models for its shared data. For this purpose the logical address space is divided into memory regions. Every region is under control of one consistency model and has a size of 512 GB, so that the upper 25 address bits specify the consistency, which is equivalent with the upper hierarchical level of page tables in our hardware (hardware details are shown in chapter IV). This design offers a fast classification of the used consistency at runtime. As default consistency Rainbow OS uses the Transactional Consistency, which covers the distributed kernel, the shared system functions and all user applications and their data.

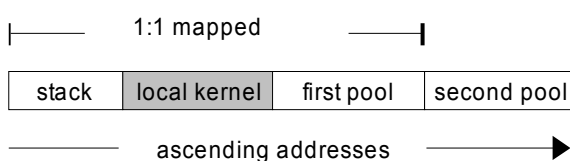


Figure 2: Local memory management

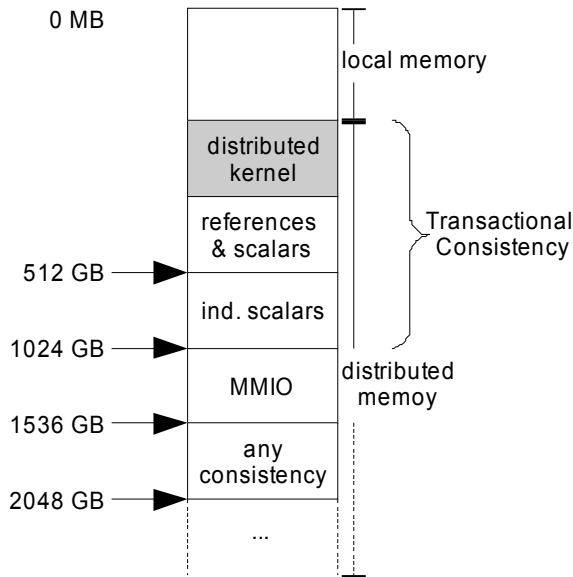


Figure 4: Distributed memory management

The so-called allocators manage the allocation of memory and object creation in their corresponding memory region. These allocators are objects itself, spanning a certain address region inside of a memory region and are subdivided into two parts (see figure 5). One part contains new objects with their references as well as direct scalars and the other contains all indirect scalars. So each memory allocation reduces the capacity of the selected allocator until its capacity is exhausted or the allocator object is cleared in memory. All object references are generally under the control of Transactional Consistency to guarantee the safety of the type system. So the first part of every allocator has to be always created in a memory region being under control of Transactional Consistency. For the second part programmers are able to use the consistency model they need. Thus, Rainbow OS features the possibility for programmers to use weaker consistency models for noncritical data without endangering the system. In future we will research how we can relax this requirement without putting at risk the runtime system. For exclusive memory usage, programmers of applications or system tools can create their own allocator avoiding potential collisions between different applications and their objects (e.g. false sharing).

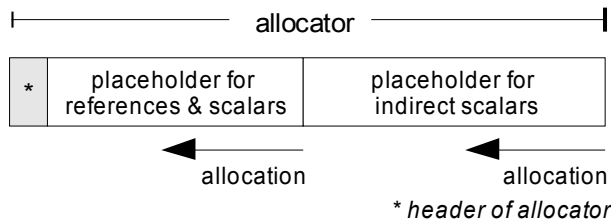


Figure 5: Allocator object structure

D. Transactional Consistency

In contrast to the well known processes of other operating systems, in Rainbow OS all computations are encapsulated in transactions. The results of a transaction are written to memory and the original state of a page is preserved by a so-called shadow copy. In case of an abort all shadow copies are restored and the transaction will be restarted automatically. At the end of a transaction the addresses of their written pages are sent to all nodes of the cluster (so-called write-set). So collisions between competing transactions can be detected by its nodes. A concurrent transaction which has read one or more modified pages have to be restarted. The set of all pages read or written is noted by the local memory management and requires marginal software action. Rainbow OS needs no locks for accessing memory or shared objects during executions. It uses an optimistic synchronization, allowing transactions to proceed locally and its validation at the end of it. [2][6][7]

III. MULTICONSISTENCY

Rainbow OS supports several consistency models, which can be implemented by kernel or system programmers. Each consistency covers a separate logical address space, which has a size of 512 GB. Rainbow OS has per default three consistency models: a local, a transactional and one special for memory mapped IO, which can be used by device drivers to map their device memory. Transactional Consistency is preset for all references in Rainbow OS, inhibiting invalid manipulations of the kernel runtime structures of Rainbow OS or the type system of it. Further we will examine using weaker consistency models for references, without putting system functions at risk. Rainbow OS is aware of every memory access and delegates it to the corresponding consistency protocol, which can operate according to its requirements. This facilitates the implementation of new consistency models in Rainbow OS without complex adaptations in system functions.

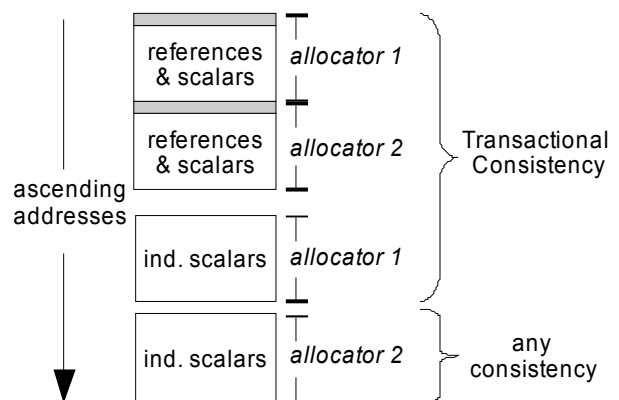


Figure 6: Allocators with different consistency models

IV. PERFORMANCE ISSUES

The measurements were performed with a cluster node, configured as following:

- processor: AMD Athlon(tm) 64 X2 Dual Core Processor 4200+
- cpu MHz: 2199.754
- motherboard: MSI K8T NEO2 V2.0
- memory: 1024 MB DDR2-RAM

Memory allocation is one of the main function of every memory management. In the context of Rainbow OS, memory management includes memory allocation and object creation, too. So we have measured the time to allocate memory and the creation of new objects, including all procedures for a new() in Java. We compared our results with the same procedures implemented in Sun Java on the same hardware. The following overview shows a selection of our results, listing the time to create a simple object, which contains no additional fields or references:

new() in local kernel (mapped memory pool)	85 ns
new() in local kernel (not mapped memory pool)	162 ns
new() in distributed kernel	321 ns
new() in Sun Java	1158 ns

Table 1: Measurements of memory allocation and object creation

The Measurements of a new() in Sun Java include probably some additional functionality inside the Java Virtual Machine like automatic garbage collection, etc.. However, our object creation and memory allocation includes some further functionality with regard to distribution and consistency, too. Furthermore our memory management has the potential for some optimizations, which will reduce the average time of a new() in Rainbow OS.

V. GARBAGE COLLECTION ISSUES

One of our next challenges will be an efficient implementation of a garbage collection, working parallel over all nodes in the cluster. Hence, the memory management has to note every creation of objects with its reference in the according object descriptor, which is part of the system informations. So all descriptors will contain a reference to a special container with a special consistency, the so-called relaxed backpacks, which include references of all instances of a certain object. On the basis of this implementation objects are traceable in memory. Garbage collections can be implemented in several ways. One way uses the type information of Rainbow OS, to identify relations and dependencies for an inspected object. So we have the possibility to analyze all kinds of references to any object. With this information it is determinable

whether an object is garbage or not. Another possibility to detect garbage is using a offline garbage collection, which is implemented in a special cluster-PC (so-called page server [3]). Due to possible collisions during the access to the relaxed backpacks, we will store these in a specific consistency model conforming with a reduced semantic, which guarantees the same content for each node but in possibly different order.

VI. PERSPECTIVES

In addition to an efficiently working garbage collection for shared memory we will design and implement several consistency models for Rainbow OS, supporting system and application programmers and their implementations to achieve best runtime performance for them. Transactional Consistency has been the strongest consistency model in Rainbow OS, it guarantees the safety of the type system, so no one can break it (in malicious intention or unintended) and endanger the runtime system. For this reason weaker consistency models are only available for primitive data types. Further we will examine the impact on the performance when using weaker consistency models for indirect scalars of objects. Furthermore in some cases it can be advantageous to relax the strong consistency restrictions for references. It could be useful when references stored in a consistency area with a weaker consistency as Transactional Consistency. But the type system of Rainbow OS has to be protected and so we will focus on how we could implement weaker consistency for references and in which cases it is useful and uncritical.

REFERENCES

- [1] Rainbow OS Homepage: www.rainbow-os.net
- [2] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, and P. Schulthess. "Optimistic synchronization and transactional consistency" *In Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 331–331, 21–24 May 2002.
- [3] S.Frenz, Reliable distributed memory with transactional consistency, Ph.D. thesis, Ulm University, 2006 (in German)
- [4] S. Frenz. Small java compiler. www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/compiler.html, 2008.
- [5] David Mosberger. Memory consistency models SIGOPS Operating Systems Review , Volume 27 Issue 1, 1993.
- [6] S. Traub, "Memory management and collision handling at transactional distributed operating systems". Phd thesis, University of Ulm, 1996
- [7] M. Wende, "Communication model of a distributed virtual memory", Ph.D. thesis, Ulm University, 2003 (in German)