

Split Objects For Multiconsistent Shared Memory

Nico Kaemmer, Patrick Schmidt, Steffen Gerhold, Thilo Schmitt, Peter Schulthess

Institute of Distributed Systems
Ulm University
Ulm, Germany
e-mail: nico.kaemmer@uni-ulm.de

Abstract—Distributed systems with shared memory and more than one consistency model for shared data are often restricted in use or inflexible for programmers. This paper describes details of our transactional distributed memory system, that provides several consistency models for shared memory. To this end Rainbow OS implements so-called split objects guaranteeing the integrity of heap structures and providing several consistency models for data.

Keywords- *transactional distributed memory, split object, multiconsistency, memory management, distributed operating system*

I. INTRODUCTION

Distributed systems with shared memory offer to the programmers an easy to use foundation for their concurrent applications. The distributed memory system allows transparent access to distributed data independent of their physical location across different cluster nodes while hiding the details of interprocess communication. Programmers need not amplify their code with explicit message passing interfaces for cluster communication and do not need a special compiler substituting code during compilation of particular communication patterns. In order to reduce the implicit communication overhead and possible synchronization latencies most cluster systems with shared memory relax their consistency constraints to improve runtime performance. Relaxing consistency assigns responsibility for correctness of system and data to programmers of kernel and applications. Our distributed operating system Rainbow OS [1] uses transactional consistency [3] for distributed code and data to guarantee a correct view of shared data. In Rainbow OS programmers can implement and use weaker consistency models for their data, without putting the type system and cluster operating system at risk. Available consistency models for data may be exchanged for customized consistency models gaining best runtime performance.

In the next chapter we will give a short outline of Rainbow OS and its features. The following chapters describe our object design and its allocation in memory. Then the management of the coexisting consistency models in our system are presented.

II. RAINBOW OS OVERVIEW

Rainbow OS is a transactional distributed memory operating systems for PC-clusters, designed for 64-bit multi-core architecture. It is almost completely implemented in a Java-like language and compiled into native code with our proprietary high-speed compiler [4]. So Rainbow OS offers system and application programmers the benefits of a type safe language coupled with fast runtime performance. Rainbow OS is partitioned into a local and a distributed part. The local part is not shared and exclusively used by an individual cluster node and includes the so-called local kernel. Object design and the details of local memory management are described in the next two chapters. Separate from the local kernel the distributed kernel entirely resides in shared memory and provides OS functions which operate on the logical perspective of the shared memory. Code and data structures of the distributed kernel are accessible and shared by all nodes of the cluster. In order to guarantee a consistent memory perspective for operating system components and application tasks Rainbow OS implements a unique transactional consistency mechanism. Before a new node can join the cluster its local kernel initializes a minimal set of drivers and a small subset of system tools. Driver instances and memory management of the local kernel are exclusive for each node and not shared in the cluster.

After the initialization of the local kernel the distributed kernel, which is fully distributed and shared by all nodes, will either be loaded from the shared memory of the cluster or it can be loaded from a separate storage medium. Distribution of the kernel implies coordinated memory management for all nodes in the cluster. Each node has the same view of the shared memory and uses the same code and data structures of the distributed kernel. In the following two chapters structure and allocation of distributed objects will be described. To guarantee a correct functionality of the cluster the distributed kernel and its data are governed by the basic transactional consistency scheme.

III. OBJECT DESIGN

For memory management purposes objects are only containers with fields of primitive data types (so-called scalars) and a set of references to other objects. As mentioned before, Rainbow OS has a local and a distributed

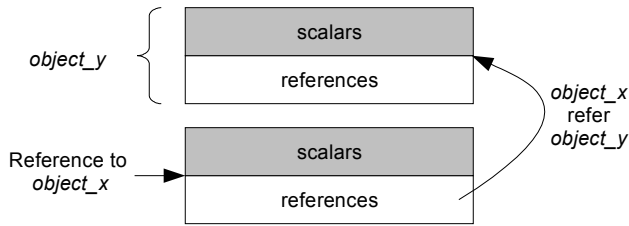


Figure 1: Local object

kernel. Both kernels contain a memory management unit each with slightly different object design.

A. Local Objects

The object layout of the local kernel differs from that of the distributed kernel. In the local kernel objects are not distributed and exclusive for each node. Local objects can be separated into two parts, one for primitive data types (so-called scalars) and one for references. The reference address to an object separates these two parts, scalar elements are stored above the object address and references below (see fig. 1).

B. Distributed Objects

Distributed objects (split objects typically) of the distributed kernel are shared between all nodes of the cluster and can be under the control of different consistency managers. To guarantee the integrity of heap structures of Rainbow OS all references to objects are kept transactionally consistent. Since the references are always kept in object descriptors all descriptors are allocated in a transactionally consistent memory region. Optionally also the scalars can be kept transactionally consistent (so-called direct scalars). A special innovation of Rainbow objects is the option to store so-called indirect scalars in a separate memory pool, outside

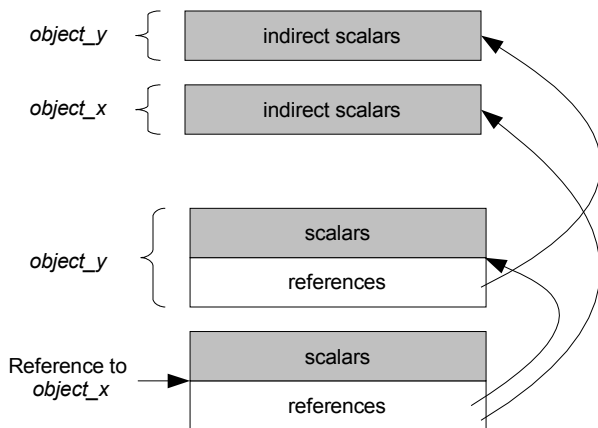


Figure 2: Split objects

of the proper object descriptor (see fig. 2). So indirect scalars often reside in a memory region which is subject to a different consistency regime. Programmers may opt to use weaker consistency models for scalar variables and scalar array elements residing in this memory region. It is the responsibility of the programmer to maintain the consistency of such weak consistency memory regions because in such a case, consistency is no longer guaranteed by the operating system but depends on proper synchronization by the programmer. Split objects are technique to retain the integrity of the type system and still to offer relaxed consistency models for data of system and applications.

IV. OBJECT ALLOCATION

Rainbow OS manages local and distributed storage resources separately. In both, the local and distributed kernel, objects are allocated using the Java operator `new()`. So-called allocator objects manage the allocation of memory and object creation.

A. Local Allocation

The memory layout of local objects is not split in memory, therefore references and scalars of local objects are stored as one descriptor and an allocator object spans only one address region in memory. The local kernel offers two different allocator objects for object allocation, that can be used by kernel and driver programmers. One of the allocators is directly mapped into memory (virtual=real), so that drivers can allocate their data structures in a straight forward manner – without converting virtual addresses to real ones. The other allocator maps memory only on demand, occupying memory if it is required. Fig. 3 shows both allocators in local memory after allocation of a few objects. Objects O1 and O2 have been allocated with the direct mapped allocator and objects O3 and O4 have been allocated with the second allocator of the local memory management.

B. Distributed Allocation

Distributed objects are split during allocation and stored at two different memory locations. Correspondingly allocator objects of the distributed kernel are partitioned into two

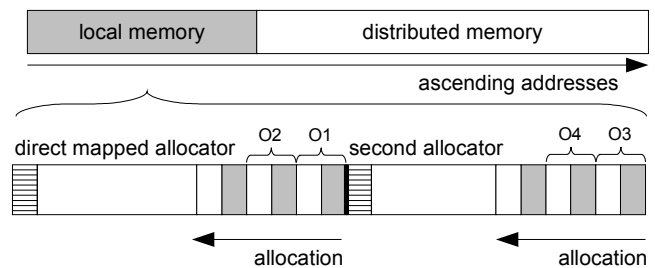


Figure 3: Local allocation of objects

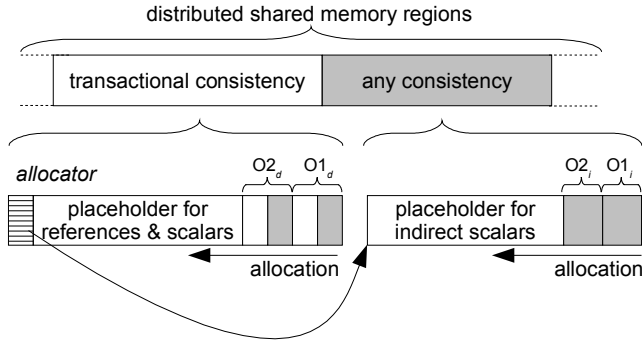


Figure 4: Allocation of split objects

parts. One part stores references and direct scalars and the other one stores all indirect scalars. The two parts of an allocator span different memory regions, which can be under control of different consistencies. The first part containing references and direct scalars is always transactionally consistent. In fig. 4 two distributed objects ($O1$ and $O2$) have been allocated. References and direct scalars of both objects ($O2_d$ and $O1_d$) have been stored in the first part of the allocator and their indirect scalars ($O2_i$ and $O1_i$) have been stored in the second part of the allocator. For explicit management of the memory model a programmer can create his own allocator with weaker consistency for its indirect scalars thus avoiding potential collisions between different nodes and their objects (due to e.g. false sharing).

V. MULTICONSISTENCY

In contrast to other known distributed shared memory cluster systems, which offer only a few consistency models for their applications, Rainbow OS can support an multitude of consistency models. Rainbow OS therefore provides a convenient platform for research and testing alternative and simultaneous consistency models in a cluster system. Optionally it is possible to change the consistency model for objects at runtime. So it is possible to adapt the consistency attributes of objects according to the changing consistency requirements of a running application.

Rainbow OS implements a separated distributed shared memory region for each consistency model. Each region covers a separate logical address range of 512 GB (see fig. 5). The first memory region is unshared memory with local

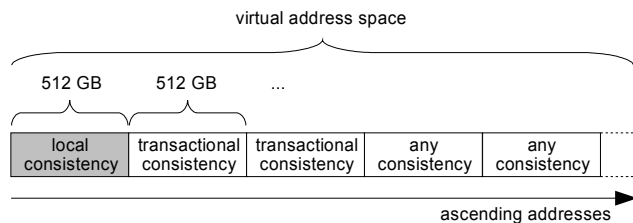


Figure 5: Virtual address space separated into memory regions

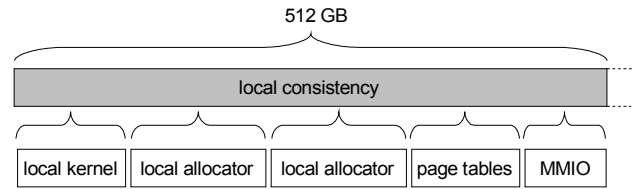


Figure 6: Local memory region under control of local consistency

consistency, followed by memory regions under control of transactional consistency and other consistency models.

A. Local Consistency

Each node in our cluster has a local memory region, that is exclusively accessible for this node and comparable to the memory of a single computer. No data in this memory region is shared between other nodes, so we have a local consistency for this memory region. This memory region is used by Rainbow OS to store the local kernel and the page tables of each node, to buffer data for drivers (e.g. network), to manage communication protocols, to map memory of devices (e.g. a graphics adapter) and so on (see fig. 6). Programmers of the local kernel can also select different allocators in this memory region.

B. Transactional Consistency

Rainbow OS uses transactional tasks for all computations, replacing the traditional process concept. Tasks share the same address space, are restartable and their data may be distributed across all nodes in the cluster (see fig. 7). Whenever a task writes the results of a computation into memory a shadow copy of the original page is kept to restore the initial state in case of an access collision with other tasks, which have manipulated the same data. In case of a collision, the aborted tasks are automatically restarted. At the end of a task the addresses of all pages written are broadcast to all other nodes. With this broadcast information the read-write or write-write conflicts between competing tasks can be detected by participating nodes. Tasks which have collided on one or more pages will be restarted. While a task is running all of its accessed pages are recorded by local memory management unit hardware. In case of collisions between tasks we use an optimistic synchronization,

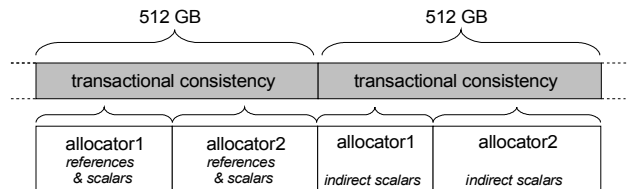


Figure 7: Distributed memory region under control of transactional consistency

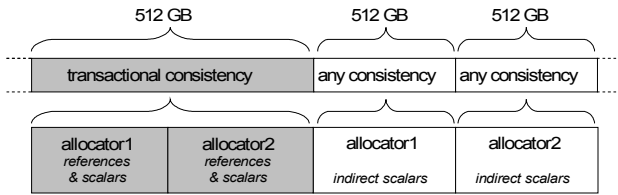


Figure 8: Memory regions under control of different consistency models

allowing tasks to proceed locally and to do delayed validation at completion time. [2][5][6]

C. Further Consistencies

Beyond local and transactional consistency, Rainbow OS can host several consistency models of different degrees of relaxation at the same time. The partitioning of the logical address space provides for each consistency model its own memory region (see fig. 8). The programmers can then define their own consistency models, suitable for the non-standard requirements of their applications. As mentioned before, weaker consistency models are only available for indirect scalars, references are always stored in a memory region under control of a transactional consistency manager.

VI. FUTURE WORK

In future we will design and implement a repertory of alternative consistency models for Rainbow OS, supporting system and applications programmers and their

implementations to reduce the probability of collisions and to achieve maximal runtime performance.

Furthermore we will examine and test weaker consistency models for some system components to optimize their performance. For example our garbage collection can use a special consistency model to collect heap and runtime information without constraining any tasks and their work in cluster being under control of transactional consistency. Another example could be the implementation of fairness strategies using a weaker consistency model to compute the best scheduling of colliding tasks, avoiding further collisions.

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] S. Traub, "Memory management and collision handling at transactional distributed operating systems". Phd thesis, University of Ulm, 1996
- [6] M. Wende, "Communcation model of a distributed virtual memory", Ph.D. thesis, Ulm University, 2003 (in German)