Transactional Distributed 64-Bit Memory for PC-Clusters

Nico Kaemmer · Steffen Gerhold · Patrick Schmidt · Michael Sonnenfroh · Stefan Frenz · Peter Schulthess

Abstract The Rainbow operating system provides 64-Bit transactional memory operation for PC-clusters. Basic consistency of the distributed objects is guaranteed by an optimistic transactions scheme and weakened consistency models are available for application data structures. The Java-like language environment allows for binary isolation, compiler-based OS security and for clusterwide garbage collection. An optional page-server will offer orthogonal persistence as well as sub-second restart and recovery.

Keywords Transactional consistency, shared memory, cluster operating system, compiler-based security, lean systems technology

Nico Kaemmer
Ulm University
Distributed Systems
E-mail: nico.kaemmer@uni-ulm.de

Steffen Gerhold
Ulm University
Distributed Systems
E-mail: steffen.gerhold@uni-ulm.de

Patrick Schmidt
Ulm University
Distributed Systems
E-mail: patrick.schmidt@uni-ulm.de

Michael Sonnenfroh
Duesseldorf University
Operating Systems
E-mail: michael.sonnenfroh@uni-ulm.de

Stefan Frenz
Ulm University
E-mail: stefan.frenz@uni-ulm.de

Peter Schulthess
Ulm University
Distributed Systems
E-mail: peter.schulthess@uni-ulm.de

1 Introduction

We present the design concepts and preliminary implementation results for the Rainbow operating system. Rainbow is based on earlier distributed shared memory operating systems [13]. It adds 64-Bit addressing, multiple consistency models, a range of garbage collection options, binary isolation and a layered security architecture. Our article is divided in the following sections:

- Rainbow OS architecture
- Transactional distributed memory management
- Reliability and persistence
- Tools and Implementations
- Perspective
- Related work

2 Rainbow OS architecture

Rainbow is a *distributed operating system* for a group of clustered PCs. To simplify programming and to avoid marshalling overhead distributed objects are allocated in a very large memory partition shared by all nodes. Figure 1 shows the partitioning of storage into a distributed memory partition and a local storage pool to host the node specific objects, such as network drivers, device buffers and page tables.

Only kernel programmers will have immediate access to core objects. But the methods and classes associated with these objects might well reside in shared memory. In the case of a multi-core CPU each core will have a separate set of core objects. Objects are allocated according to their sharing characteristics whereas their access privileges are governed by the compiler and the associated class and package hierarchy.

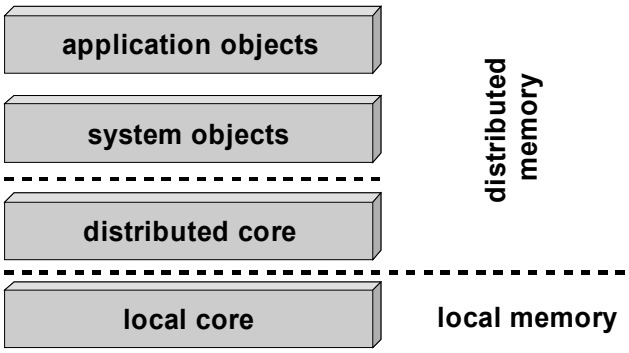

Fig. 1 Layers of Rainbow OS

Kernel programmers are allowed to use the pseudo-class MAGIC which gives them access to all local hardware. They are not constrained by the type system of the implementation language. To make informal verification easy and to reduce the risk of errors the code size of the kernel is kept to a minimum.

System programmers will implement mission critical algorithms both at the level of the local station and at the global cluster level. Algorithms at this level include logical storage management, clusterwide name service, insertion and deletion of transactions and high-level device control. However, they will access hardware and devices only through a formal kernel interface class. System-, kernel- and lowlevel device-programmers carry full responsability for the correct operation of the cluster, but to a certain extent they are protected from their own errors by the language type system.

Application programmers work within a Java sandbox and under no condtion they should be able to bring down the local node or the cluster. The sandbox is enforced even though the compiler will generate machine code instead of byte code. At some later stage of development a quota system will be introduced for application data storage.

The concept of *binary isolation* prevents the introduction of manipulated binary code. There is no facility to import binary code segments or object reference patterns into a running Rainbow cluster. Only the compiler is privileged to generate code segments for classes and method. As a special privilege Kernel code segments may use MAGIC and are only created if the invoking programmer has kernel privileges.

3 Transactional distributed memory management

Distributed memory management is a basic component of any distributed operation system, providing extended functionality compared to single system memory management. Rainbow OS essentially has two components for memory allocation, one for the local memory, integrated in the local core of Rainbow OS, and one for the the distributed memory, as part of the distributed core, which is shared by all nodes. The local memory management administrates the available physical memory, the page tables for virtual memory and offers a simplified memory allocation function for all objects of the local core. The distributed core of Rainbow OS contains an additional sophisticated separate memory management for shared cluster objects and multiple consistency models.

3.1 Local Memory Management

Memory management of the local core offers two memory pools for object allocation: a directly mapped pool, in which corresponding physical and logical addresses are equal and a normal pool, that maps physical pages on demand. Both pools are placed logically between the local and the distributed core and can be used by drivers and system functions to allocate local objects.

3.1.1 Object Design and Allocation

The interface for object allocation for developers or programmers is the well-known invocation of New in Java, which is mapped by the compiler [6] to a method call to the runtime environment of Rainbow OS. This method allocates a corresponding area of memory and initializes the object itself. The internal object structure in Rainbow OS deviates from the Sun JVM object structures. Rainbow OS and the compiler divide objects divided in two parts, one containing the references and the other the scalars. References to object are pointing to the separation line between the two parts, whilst references to other objects are stored below and scalars above this pointer (see Figure 2). In contrast to other operating systems and Sun Java, Rainbow OS offers the possibility to store scalars of an object at a distant position in memory. These scalars are called indirect scalars and are managed with an additional reference in the corresponding object. This potential separation of object scalars allows Rainbow OS to support multiple and weakened consistency models without putting the type system at risk. In the local memory management of Rainbow OS these indirect scalars are unused, because the local core and its objects are locally consistent and have to be as simple as possible avoiding unnecessary complexity and extensive structures.

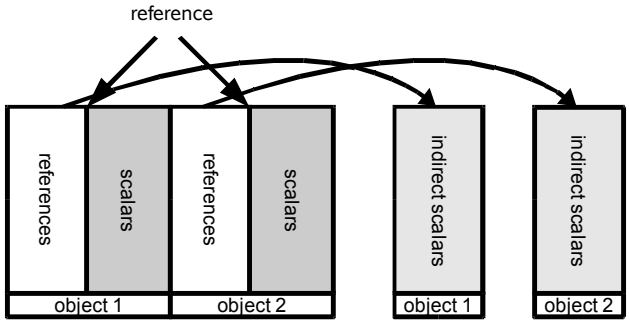

Fig. 2 Object structure

3.1.2 Local Consistency

The local core of Rainbow OS and its objects are subject to a simple local consistency model comparable to a normal singleuser system. Any read to a memory location returns the most recent write operation to this location. No other node in the cluster can manipulate or access this part of local memory, it is not shared and only accessible to kernel or system programmers.

3.2 Distributed Memory Management

As mentioned earlier Rainbow OS is divided in a local and a distributed memory, with separate management components. The distributed memory management includes all shared memory pools (so-called allocators) and several consistency models. Due to the existence of two cores, a local and a distributed one, and the fact that not every driver or system component can be shared by all nodes, an additional communication facility between both cores is necessary. Therefore Rainbow OS offers a simple Integer- Interface with proprietary data buffers to communicate between the distributed and the local core widening the scope in which code can be shared. Drivers for example can be implemented in the distributed core of Rainbow OS too.

3.2.1 Logical Address Space

Rainbow OS identifies and separates several consistency models by inspecting their logical addresses. Each consistency pool spans a logical address space 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 (AMD64). This implementation offers a fast consistency classification at runtime, without complex lookup operations or algorithms. By default Rainbow OS uses a Transactional Consistency Model [13] for distributed core, shared system functions and all user applications. By default object references are under the control of Transactional Consistency to guarantee the safety of the type system. Programmers may optionally use a weaker consistency only for scalars variables and arrays. Future research will show how to relax this requirement without putting the runtime system at risk.

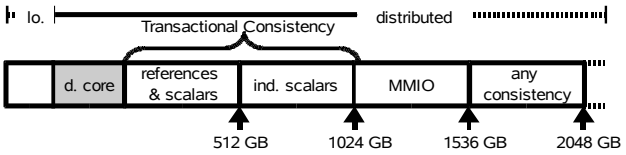

Fig. 3 Logical address space

3.2.2 Object Design and Object Allocation

Inside a consistency pool so-called allocators manage the memory allocation and object creation. These allocators are objects themselves and properly partitionned into direct and indirect object fields (see Figure 4). A descriptor part contains references and direct scalars and the other one contains all indirect scalars. For explicit management of the memory model a programmer can create his own allocator with weaker consistency thus avoiding potential collisions between different applications and their objects.

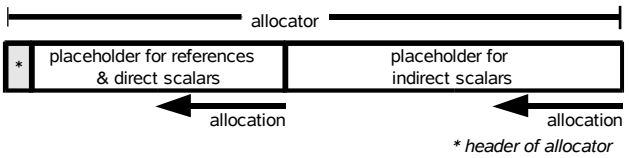

Fig. 4 Structure of an allocator object

3.3 Multi Consistency

Rainbow OS supports several consistency models, which have been implemented by kernel or system programmers. Each consistency covers a separate logical address space, which has a size of 512 GB. Rainbow OS offer three consistency models by default: local, transactional and one special local for MMIO, which can be used by device drivers to map their device memory. Transactional Consistency is preset for all references in Rainbow OS, precluding invalid manipulations of the kernel runtime structures or of the type system. In the future we will investigate, to what extent we can use 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 then operate according to its requirements.

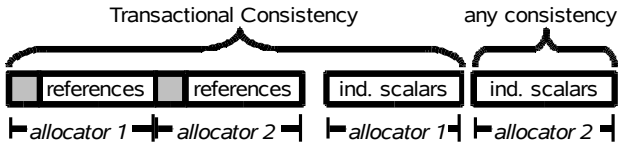

Fig. 5 Allocators representing different consistencies

4 Reliability and persistence

4.1 Introduction to Checkpointing

Checkpointing is a common way of providing fault tolerance and reliability in a distributed system. If a long-running application is checkpointed from time to time, a possible runtime error or hardware fault does not force the application to start over from the beginning as it can fall back to a recent checkpoint and resume from that point.

Traditional operating systems like Linux or Microsoft Windows find it difficult to provide a transparent application checkpointing mechanism as the entire state of a running application consist of a user context and a system context (i.e. open file descriptors or process IDs). A checkpointing facility running solely in userspace might be able to read the user context but is usually forbidden to read the system context let alone to modify it in case of a fallback. A kernel mode checkpointing facility on the other hand can access both the user contexts and system contexts, thus being able to form consistent checkpoints of an application and to support restoring all changed structures in case of a fallback.

The major disadvantage of a kernel mode checkpointing is the need to adapt it to all release changes in the OS kernel, be it a security patch or continuous development. The checkpointing task becomes even more difficult and complex if the application is not run locally on only one computer but is distributed to many nodes in a computer network. In contrast to local execution the checkpointing facility must now also consider a higher degree of both parallelism and message latency throughout the network in order to guarantee consistent snapshots of distributed applications.

4.2 Smart Checkpointing

A transactional distributed memory (TDM) system reduces the effort for consistent distributed snapshots to a minimum by taking advantage of the transactional character of its distributed memory as described in section 3. These "Smart Snapshots" make it possible to avoid the drawbacks mentioned above and gain the capability of taking snapshots of the entire operating system at once which obviously includes all running applications. In order to create a checkpoint of an entire distributed system it is usually not sufficient to have every system node store its own memory content independently from other nodes because of potential internode dependencies thus requiring additional coordination methods. Unlike traditional distributed memories Rainbow OS implements a transactional distributed memory which easily produces consistent memory images of the entire cluster. This simplifies the task of storing a consistent snapshot: It suffices to read out the entire system image and to store it on a persistent device (e.g. a hard disk). Since this task can be performed using existing methods provided by the transactional consistency module, taking a checkpoint adds little additional complexity to the system. No auxiliary algorithms for distributed snapshots are required.

4.3 Checkpointing in Rainbow OS

The major part of the Rainbow kernel runs in a transactionally distributed mode in conjunction with the application transactions. Since the security layers mentioned in section 1 prevent unintended or malicious access from the applications to the kernel, no address space separation is required. Smart Checkpoints of the entire Rainbow cluster are easily constructed by storing this global TDM image to a persistent device. In case of a fallback there is no need to specifically read and modify application-relevant kernel structures. Nearly all important kernel components such as the distributed memory management, the task scheduler and most drivers live within the transactional distributed memory and are therefore automatically and transparently included in every system checkpoint. Those modules of the Rainbow kernel which run locally contain only very few information items which are relevant to distributed kernel components or applications. One example of a locally executed kernel feature are the modules providing the transactional consistence and the network access. Both of them can be restarted without losing critical information which is not also represented by a distributed object and thus included in any checkpoint.

As can be seen from the discussion above, using a TDM system such as Rainbow OS which keeps most of its kernel inside the TDM provides an elegant and easy way to overcome consistency issues on a single computer as well as on an entire network cluster. As the transactional approach introduces only a small additional overhead it opens up the possibility to create system-wide checkpoints with a frequency of a few seconds [7].

4.4 Performance issues

In order to implement a high-performance checkpointing facility it is important to consider typical run-time scenarios. In the Rainbow pageserver two different situations can occur: in normal cluster operation mode the pageserver is put in charge of regularly creating checkpoints of the TDM which results in heavy disk write operations and only very few read accesses. If a fatal error induces the fallback of some cluster nodes, the page server is then faced with different requirements: the cluster nodes affected by the fatal error are now requesting the consistent version of TDM data which they lost due to their fallback. The page server is now mainly answering data page requests from its hard disk. A highperformace page server must therefore implement an efficient writing strategy as well as a technique to quickly locate and access stored data on its disk. Our previous 32-bit page server implements a special disk access algorithm called "linear segment" [5] which was specially designed to meet the performance expectations in a transactional cluster environment. It gains maximum write performance on hard disks by writing in a strict linear fashion, thus reducing disk seek operations to an absolute minimum.

4.5 Advancing to 64 bit architecture

Porting the Rainbow page server from 32 bit to 64 bit introduces interesting challenges regarding the virtual address space size. The available virtual address space increases by more than four orders of magnitude from 4 Gigabyte to 256 Terabyte[1] while the installed physical memory is solely enlarged by factor 8 from 1 Gigabyte to 8 Gigabyte. This increase in address space calls for an adaptation of formerly applied checkpoint storing implementations. As depicted in figure 6, a Rainbow snapshot basically consists of two different kinds of information: the image data which includes all content of the transactional distributed memory and the meta data which among other things describes the mappings between each stored page and its respective block address on the hard disk. In the 32 bit version of Rainbow each checkpoint contains an exhaustive list which specified the hard disk block address of every logical page of the TDM.

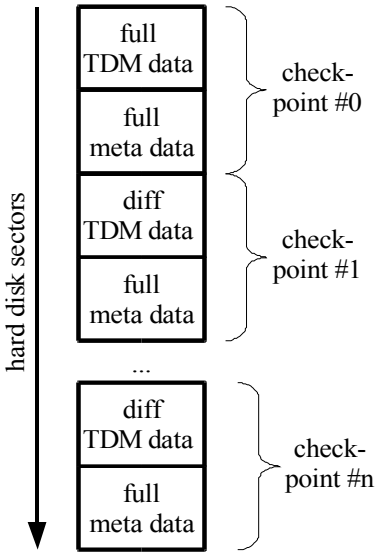

Fig. 6 Page server disc structure (32 bit)

Due to the vast virtual address space of a 64-bit system this is not a viable solution anymore as a complete list of all pages would contain 2^36 entries. The amount of meta data per snapshot can be reduced by switching from a full page list to an incremental list which only contains modified pages, but this optimization poses additional challenges as well since it makes the task of finding a specific version of a TDM page much more difficult. In the 32 bit system the hard disk block address of any TDM page stored at any given logical time can be easily detected by parsing the corresponding meta data structure. This direct approach is not feasible anymore if an incremental representation of meta data is used. Challenge: find new algorithm to provide fast write access, short read seek times, low memory consumption, acceptable complexity of disk reordering.

4.6 Advancing to solid state disks

As described in subsection 4.4 an optimal algorithm must combine high write throughput of page data with fast read access in case of a fallback. Common magnetic storage devices such as hard disks perform best if the seek time between two subsequent commands is minimized. During normal cluster operation many pages

[1] Despite the notion, the virtual address space width of current AMD64 / Intel64 architectures is only 48 bit.

are written to the pageservers hard disk aproximately in the order in which the modifications occur. In case of fallback the cluster nodes most probably read those pages in a completely different order. As can be seen there is no perfect sequence of pages which suits maximum write throughput as well as maximum read performance when using magentic hard disks. The current rise of solid state disk (SSD) drives provides an interesting opportunity to tackle this performance paradoxon. Since SSD drives use flash chips to store information their access times are potentially low compared to magnetic disk drives because no penalty for positioning the hard disk head is involved anymore. Additionally SSDs show great promise for massively increasing the absolute read and write performance by spreading the workload to many flash chips which work in a parallel. Todays SSDs such as Intels X25-E [8] already feature high performance combined with low access latencies which makes them the persistent storage media of choice for the Rainbow page server. Switching from hard disk drives to SDD drives requires fundamental changes in the disk storage stuctures and the corresponding algorithms. On the one hand it is not necessary any more to access the device in a strictly linear way to achieve acceptable write performance. On the other hand it is important to consider the specific peculiarities of flash media chips and the internal composition of SSDs. In contrast to hard disk drives SDDs feature different optimal block sizes for read and write accesses. As can be seen in Figure 7 it is usually possible to read small amounts of non-subsequent data such as randomly chosen 4 KB blocks in an efficient way, but writing the same blocks in the same order will result in comparatively low throughput. This is caused by a write access block granularity which is usally much bigger than 4 KB in SSDs which makes write accesses with smaller block size be mapped to more time-consuming readmodify-write sequences inside the drive. It is therefore very important to adapt the new algorithm for the 64 bit pageserver bearing these special requirements in mind: disk blocks of 4 KB or larger can be read with acceptable speed, but write access should take place with large block granularity to avoid internal throughput penalties by the SSD.

Random 4 KB Reads	35 K IOPS
Random 4 KB Writes	3.3 K IOPS
Sustained Sequential	up to 250 MB/s
Read Sustained Sequential Write	up to 170 MB/s

Fig. 7 Performance figures for Intel X25-E SATA SDD [8]

5 Tools and implementation

5.1 SJC – Small Java Compiler

The Small Java Compiler SJC is a lean but sophisticated compiler translating (a subset of the) Java language into native code for different architectures; targeting 8, 16, 32 and 64 bit processors (see [6]). The special class "MAGIC" grants low-level hardware access beyond the traditional Java sandbox. SJC thus reconciles driver & OS-level programming with the benefits of a type-safe language avoiding potential programming errors due to memory arithmetic or misused pointers (see [14]). The runtime structures created by the compiler discriminate between primitive data-types (called scalars) and reference variables. The runtime structures created by the compiler are well organized (see [6]) and simplify the inspection of references in tasks like relocation or garbage collection. There are two categories for scalar objects depending on the intended memory consistency model: "direct scalars" are directly allocated in the object descriptor and "indirect scalars" are allocated in a separate allocation pool. The compiler itself was also implemented in Java with appropriate packaging to allow the addition of a new language frontend, the provision of additional codegenerators (backend) for different target machines. The creation of disk-bootable images, hexfiles for EEPROM flashing or executable files for Linux and Microsoft Windows works smoothly due to the clean packaging of the compiler. The complete SJC tool chain (see Figure 8) from source code to output file is modularized, well arranged and easily teachable. Comprising the complete toolchain, no separate linkers or loaders are required to create an executable or bootable image.

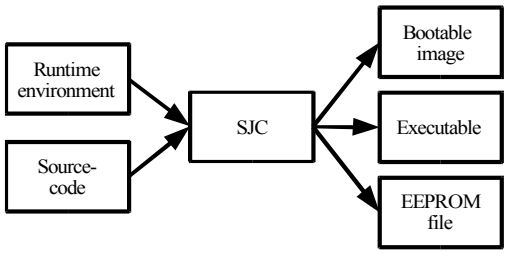

Fig. 8 Tool-chain in SJC

5.2 Binding schemes

Depending on the operation mode – static objects or movable objects – the compiler resolves references during compilation by inserting absolute addresses (static)

or taking a detour by dereferencing the class descriptor of the invoked instance (movable). Whereas the generation of movable code facilitates the relocation of data and code, the static compilation mode offers faster program execution (see Figure 9).

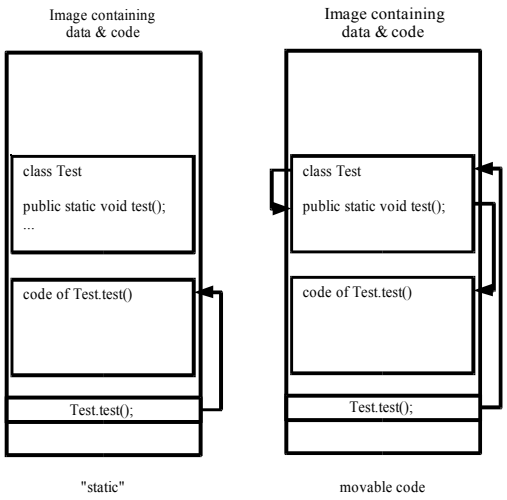

Fig. 9 Binding schemes in SJC

During (the still ongoing) enhancement and extension of the compiler, much attention was paid to its ability of selfcompilation, i.e. the compiler is capable of compiling itself. Therefore, it is possible to integrate the compiler itself in the source-code to be compiled, providing the program or system contained in the resulting image with the potential ability for evolution or extension. Consequently, Rainbow OS contains its compiler which can be used to extend the running system easily with new functionality.

Besides the advantages with respect to security and safety issues, the usage of a type-safe and widespread programming language like Java offers the opportunity to benefit from existing IDEs such as Eclipse, facilitating the development process.

5.3 Green Shell

Like other operating system, too, Rainbow will provide several possibilities for interaction with the user ranging from text-based commands to graphical user interfaces. A representative of the text-based user input is the so-called "Green shell". Its name origins from the predecessor of Rainbow OS, Green OS and the color used for displaying text. The Green shell is designed to be a lean, but fully functional user shell application. It provides several features setting it apart from other usual text-based inputinterfaces such as the possibility to append additional virtual screens to the current screen displayed on the monitor and rotate between these screens and the concept of *active text*.

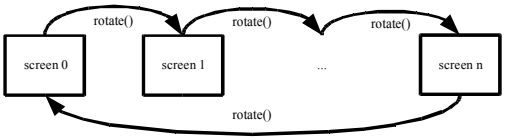

Fig. 10 Rotating between virtual screens

Active text takes advantage of the fact that in most cases the commands available for invocation are already displayed on the monitor. The user therefore can simply navigate with his cursor to the desired command and execute it with a special shortcut. For the case that a supported command isnt currently displayed on the green-shell, the user can simply type it in and invoke it in the same way. As Rainbow OS keeps all its code and data in the Transactional Distributed Memory, it can dispense with the burden of managing a file-system. Code and data is represented by objects and runtime structures in the TDM which can be invoked and accessed by *active text* calls. If, for instance, a user wants to invoke a method "run" implemented by the class "Task", he can simply click on a preexisting text piece "Task.run" to execute that method.

5.4 Wissenheim

A real application, proving the feasibility and performance of the concepts integrated in Rainbow OS is "World of Wissenheim". The "World of Wissenheim" project is an interactive 3D world intended to provide a learning and leisure environment. The representation of the participants by avatars suggests a direct form of "virtual presence" facilitating encounters in the virtual world. Approximately 30 topics, interactive animations and social scenarios have already been implemented. In contrast to classical message-passing based interactive applications Wissenheim makes use of the transactional memory paradigm to share a scene graph structure among the participating nodes. Every node can access and alter the scene graph directly allowing a more direct programming model. Wissenheim uses Rainbows unique ability to support weaker consistency models to relax consistency for highly interactive but non critical content to boost overall performance.

6 Perspective

6.1 Garbage collection issues

An interesting research topic is the evaluation of different garbage collection schemes. Currently memory management will record every new object instance and its reference variable in a so-called backpack. With this information it is determinable whether an object is garbage or not. More or less relaxed reference tracking schemes are conceivable and will be studied.

Another possibility to detect garbage is using offline garbage collection on the page server. As the pageserver holds successive versions of cluster checkpoints on its disk, it can easily perform garbage collection tasks on the stored images which are not subject to change anymore. Although those TDM snapshots are not completely up-to-date, they can be used to detect objects which are no longer reachable from the object root set.

Since the type-safe implementation of Rainbow allows for nonconservative garbage collection algorithms, it is obvious that any object which was once found to be garbage remains garbage until its memory is reclaimed by the operating system. Thus it is possible to supplement the on-line garbage collection algorithm which runs in a distributed fashion on some or all cluster nodes with special "hints from the past". These hints will indicate the conservative garbage state of objects, in spite of the fact that this information was collected "off-line" on older TDM checkpoint images.

6.2 Minimally-invasive recompilation

Improvement, extension and evolution have always been characteristics of software development. Whereas these evolution issues are easily solved for stand-alone, short-running software, a long-running system with a persistent memory imposes several challenges. Considering the recompilation of existing classes in a system, the literature distinguishes between several stages ranging from the so-called "big bang" recompilation (simply recompile all existing classes) over cutoff-recompilation (limiting the amount of recompiled classes by observing dependencies) up to "smart" recompilation techniques [1]. Compatibility of runtime structures, consistency and type-safety concerns are of special interest as they directly interact with the possibilities and limits of the applicable recompilation modes.

Another issue is the evolution of instances referencing obsolete versions of classes. Due to this fact, the limitation of the amount of recompiled classes is of high interest as it directly affects the number of instances to be converted, which may add up to a considerable amount in an orthogonally persistent operating system. Additionally the use of a cluster operating system with transactional and other consistency models brings the efficient distribution of the work load for recompilation and evolution tasks into focus. All these issues are part of the work done on the minimally invasive recompilation techniques in the Rainbow cluster operating system with the perspective of an operating system, extensible and evolvable at runtime with a minimal impact on the runtime structures affected by the changes, and yet providing the ease of a type-safe programming language.

7 Related work

L. Keedy presented the idea of distributed shared memory systems in 1985 [9]. IVY [10], Mirage [4] and TreadMarks [3] implemented page-based distributed shared memory systems with different consistency models. A formal treatment of consistency models is available in [15], leading to references and discussions of a multitude of DSM systems and middleware packages.

The XtreemOS project aims at enabling Linux for the Grid by integrating vital Grid services directly into the operating system. The Object Sharing Service of XtreemOS [11] provides a transparent way for data sharing of distributed applications running in a grid environment using a transactional memory approach.

The Intel STM C++ compiler [2] [12] integrates several means for defining transactional regions in the code including a "High Performance Parallel Optimizer", an "Automatic Vectorizer" and a "Multi-Threaded Application Support". These compilerdirectives guarantee the proper operational sequence of the code regions executed in parallel.

References

1. R. Adams, W. Tichy, and A. Weinert. The cost of selective recompilation and environment processing. *ACM Trans. Softw. Eng. Methodol.*, 3(1):3–28, 1994.
2. A. Adl-Tabatabai, B. Lewis, V. Menon, B. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Canada*, 2006. STM.
3. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):1828, 1996.
4. B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. *SIGOPS Oper. Syst. Rev.*, 23(5):211223, 1989.
5. S. Frenz. *Zuverlaessiger verteilter Speicher mit transaktionaler Konsistenz.* PhD thesis, University of Ulm, 2006.

6. S. Frenz. Small java compiler. `www-vs.informatik.uniulm.de/dept/staff/frenz/private/compiler.html`, 2008.
7. S. Gerhold, M. Schoettner, M. Fakler, M. Sonnenfroh, and P. Schulthess. Smart snapshots on top of a distributed transactional memory. 03 2007.
8. Intel. *IntelA ®X25-E SATA Solid State Drive.* Intel Corporation.
9. J. L. Keedy and D. A. Abramson. Implementing a large virtual memory in a distributed computing system. In *In Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, 1985.
10. K. Li. Ivy: A shared virutal memory system for parallel computing. In International Conference on Parallel Processing, 1988.
11. M.-F. Mueller, K.-T. Moeller, M. Sonnenfroh, and M. Schoettner. Transactional data sharing in grids. In *Proc. of the International Conference on Parallel and Distributed Computing and Systems*, 2008.
12. B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang. Enabling scalability and performance in a large scale cmp environment. In *EuroSys 07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 7386. ACM, 2007.
13. 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*, page 331, 21-24 May 2002.
14. N. Wirth and J. Gutknecht. *Project Oberon - The Design of an Operating System and Compiler.* Number SBN 0-201-54428-8. Addison-Wesley, 1992.
15. David Mosberger. Memory consistency models. SIGOPS Operating Systems Review , Volume 27 Issue 1, 1993.