Transparent Orthogonal Checkpointing Through User-Level Pagers

Espen Skoglund, Christian Ceelen, and Jochen Liedtke

System Architecture Group
University of Karlsruhe
{skoglund,ceelen,liedtke}@ira.uka.de

Abstract. Orthogonal persistence opens up the possibility for a number of applications. We present an approach for easily enabling transparent orthogonal persistence, basically on top of a modern μ-kernel. Not only are all data objects made persistent. Threads and tasks are also treated as normal data objects, making the threads and tasks persistent between system restarts. As such, the system is fault surviving. Persistence is achieved by the means of a transparent checkpoint server running in user-level. The checkpoint server takes regular snapshots of all user-level memory in the system, and also of the thread control blocks inside the kernel. The execution of the checkpointing itself is completely transparent to the μ-kernel, and only a few recovery mechanisms need to be implemented inside the kernel in order to support checkpointing. During system recovery (after a crash or a controlled shutdown), the consistency of threads is assured by the fact that all their user-level state (user memory) and kernel-level state (thread control blocks) will reside in stable storage. All other kernel state in the system can be reconstructed either upon initial recovery, or by standard page fault mechanisms during runtime.

1 Introduction

Persistent object stores have a long history of active research in computer science. Little research, however, has been targeted at providing orthogonal persistence, and still fewer systems have actually been implemented.

Orthogonal persistence does make sense though, especially with the advent of devices like PDAs. A day planner application would for instance always be running on your machine. Since the application would be persistent, it would transparently survive power failures or system crashes. There would be no need for the application programmer to explicitly store the results of user modifications, effectively saving both development time and program size. Desktop machines and larger servers would also be able to benefit from orthogonal persistence. A long running compute-intensive application would not need to contain complicated (and very likely error-prone) checkpointing code. Researchers could concentrate on algorithm design, and leave the checkpointing to the operating system.

In this paper, we present an approach for easily implementing transparent orthogonal persistence on top of a modern μ-kernel. We hope to prove that implementing orthogonal checkpointing is not inherently difficult, even in a system that was not originally designed for that purpose. One important aspect of the design is that the kernel does not need to collect any kernel data structures into some externalized form (i.e., there is no need to perform the process known as *pickling*).

Before describing the checkpointing design (Sects. 4, 5, and 6), we give a brief overview of the μ-kernel used as a base for the implementation (Sect. 2) and our general approach (Sect. 3). Section 7 lists related work, and finally Sect. 8 concludes.

2 Implementation Basis: The L4 μ-Kernel

The L4 μ-kernel [10] is a lean second generation μ-kernel. The philosophy behind the kernel is that only a minimal set of concepts is implemented within it. A concept is permitted inside the kernel only if moving it outside the kernel would prevent some system functionality to be implemented. In other words, the kernel should not enforce any specific policy on the surrounding system. Following this philosophy, the L4 μ-kernel implements two basic user-level abstractions: threads and address spaces.[1]

2.1 Threads and IPC

A thread is an activity executing inside an address space. Several threads may exist within a single address space, and the collection of threads within an address space is often referred to as a task. Threads (and in particular threads within different address spaces) communicate with each other using inter-process communication (IPC).

The IPC primitive is one of the most fundamental μ-kernel mechanisms. Not only is it the primary means of communication between threads, it is also used as an abstraction for exceptions and interruptions. For instance, if a thread raises a page fault exception, the exception is translated into an IPC message that is sent to another thread which handles the page fault (see Fig. 1). If a hardware interruption occurs, the interrupt is translated into an IPC message and sent to a thread that handles the interruption. This latter example also illustrates an important aspect of the μ-kernel design: device drivers need not be implemented inside the kernel. They may be implemented as threads running in user-level.

[1] A rationale for the choice of these basic abstractions and a detailed description of the L4 μ-kernel is given in [10].

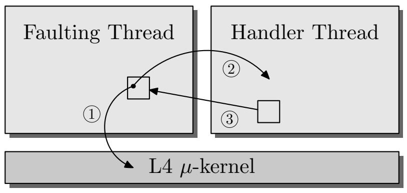


Fig. 1. Page fault IPC. When a page fault occurs in a user level thread; (1) a page fault exception is raised and caught by the μ-kernel, (2) the kernel generates a page fault IPC from the faulting thread to the handler thread, and (3) the handler thread (possibly) maps a page frame to the faulting location.

2.2 Recursive Address Spaces

Recursive address spaces forms a concept that is relied heavily upon by the checkpointing facility presented in this paper. With recursive address spaces, a task may map parts of its address space to other tasks. These other tasks may in turn map the mapped parts to other tasks, creating a tree of mappings. Upon mapping to another task though, control of the mapped part will not be relinquished. A mapper may at any time revoke its mappings, recursively rendering the mappings in other tasks invalid.

Recursive address spaces enable memory managers to be stacked upon each other. These memory managers are also referred to as pagers. This does not necessarily imply that they swap memory to stable storage (though that may often be the case), but that they handle page faults for threads in their sibling tasks. The top-level pager in the system is backed by physical memory. That is, its address space is mapped idempotently into physical memory. As such, it controls all physical memory in the machine[2] and has no need for pagers backing it.

There has been much concern as to whether μ-kernels suffer a too large performance degradation due to the added number of context switches imposed on the system. It has been argued, however, that this is mostly an artifact of poor kernel implementations, and can be alleviated by properly implemented kernel mechanisms [10].

3 General Approach

This paper focuses on how to implement non-distributed per-machine checkpoints. Cross-machine synchronization and synchronization between persistent and non-persistent objects, threads, and tasks are not topics of this paper.

More precisely, this paper discusses how to implement checkpoints *on top of a microkernel,* i.e., (a) through user-level servers and (b) relatively orthogonal (and thus independent) to other (legacy) system services such as pagers, file systems,

[2] Except for some memory which is dedicated to the μ-kernel, for example page tables.

network stacks, security servers, and so forth. Figure 2 gives an architectural overview of the system, illustrating the μ-kernel, device drivers, the checkpoint server, and the persistent pagers and applications serviced by the underlying persistence run-time system.

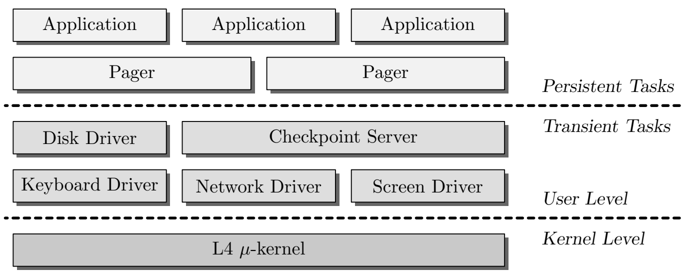


Fig. 2. Architectural overview

3.1 Transient and Explicitly-Persistent Objects

A real system always contains a mixture of transient (non-persistent) and persistent objects. Extreme examples are device drivers that typically can not be made persistent and file systems that are always persistent. So, a persistent application might "use" both persistent and non-persistent objects. Nicely, in an orthogonally persistent system, the applications' address spaces, stacks, registers, and thread states themselves are all persistent objects. Nevertheless, many orthogonally persistent applications will use or contact non-persistent objects and/or servers, e.g., a clock/date service, a remote web site, a phone line, a video camera, a gps receiver, and so on.

Connecting to non-persistent services and using non-persistent objects is a problem that strongly depends on the application and on the nature of the non-persistent servers and objects. A persistent system can only offer basic mechanisms to detect inconsistencies (i.e., the necessity of recovery) and to prevent applications from using unrecovered objects or server connections once a system restart happens or a non-persistent object or server crashes. The general system mechanism for customized user-level recovery in such cases is *to invalidate all connections (including identifiers) to non-persistent objects* once the persistent system restarts (or the non-persistent objects crash or are partitioned from the persistent system).

File systems are well-known members of the class of *explicitly persistent* servers. Such servers implement explicitly persistent objects. If a persistent application uses such servers it has either to deal with a similar synchronization problem as for non-persistent objects, or the system has to synchronize its own checkpoints and the checkpoints of all used explicitly-persistent servers, e.g., through a two-phase-commit protocol. The current paper does not discuss the corresponding problems.

3.2 Implicitly-Persistent Objects

The focus of the current paper is *implicitly persistent* servers (and applications). That is, how do we make servers and applications orthogonally persistent that originally do not include persistence, i.e., that do not include a checkpoint/recover mechanisms. Implicit persistence should be implemented by a set of system servers that *transparently* add orthogonal persistence to existing non-persistent servers and applications.

For implicit persistence we need to implement transparent checkpointing. Basically, we have to solve three problems:

Kernel Checkpointing. How can we checkpoint the microkernel in such a way that all persistent threads, address spaces, and other kernel objects can be recovered?

Main-Memory Checkpointing. How can we checkpoint those main-memory page frames that are used by persistent applications and implicitly persistent servers?

Backing-Store Checkpointing. How can we checkpoint backing store (disk) blocks that are used by implicitly persistent servers? Or in other words, how can we ensure that transactions made to stable storage by implicitly persistent servers are synchronized with their checkpoints?

4 Kernel Checkpointing

4.1 The Checkpoint Server

The checkpoint server acts as a pager for thread control blocks (TCBs) in the system. This permits it to save the kernel state of the various threads.

Giving control of the thread control blocks to the checkpointer is achieved by not having the kernel own the physical page frames of the TCBs. Instead, when the kernel needs more backing memory for TCBs (whether it be for a persistent or a transient thread), it requests the checkpointer to map a page frame to the kernel (see Fig. 3). The mapped page frame will be owned by the checkpointer, and the checkpointer may at any time access the TCB or unmap it from the kernel. Letting the checkpointer retain full control of the TCBs implies that the checkpointer—just like device drivers—must be a trusted task.

At regular intervals, the checkpointer takes a consistent copy (a *fixpoint*) of all needed system state (i.e., thread control blocks of persistent threads) and writes it to stable storage. A consistent copy could be assured by establishing some causal consistency protocol. Such a protocol, however, would require that the checkpointer has knowledge of all communication between persistent threads. This is not feasible though, since threads can communicate in ways which are

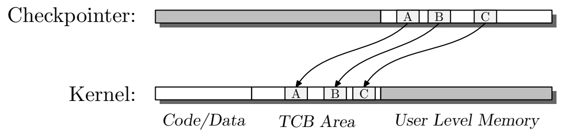
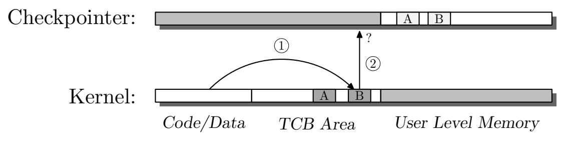
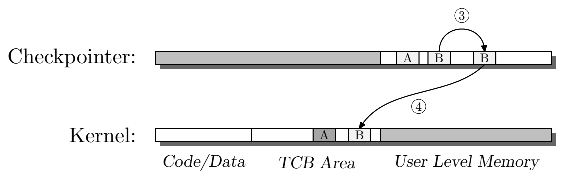


Fig. 3. TCB paging. The checkpointer owns the pages residing in the TCB area of the kernel. Arrows indicate mappings from the checkpointer's address space into the TCB area.

not known to the checkpointer, e.g., through IPC or shared memory. The checkpointer therefore implements a simple scheme, taking a snapshot of the system at a single point in time. Since the checkpointer is implemented as a regular user-level task, it is subject to interruptions by the in-kernel scheduler, or by hardware devices. This could cause other persistent tasks to be scheduled before the checkpointer has written their TCBs to stable storage, rendering the fixpoint in an inconsistent state. To cope with this problem, the checkpointer initially makes all persistent thread control blocks copy-on-write (see Fig. 4). Once this is accomplished, the checkpointer can lazily store their contents to stable storage.



(a) The TCBs in the kernel area have been marked copy-on-write (dark grey). The kernel then tries to, e.g., schedule thread B. This requires access to B's TCB (1), which in turn raises a page fault that is directed to the checkpointer (2).



(b) The checkpointer copies the accessed TCB to a new location (3), and maps the copy of the TCB to the kernel with write permissions (4). The kernel may now modify the TCB copy without disturbing the old contents.

Fig. 4. The TCB copy-on-write scheme

During the copy-on-write operation the checkpoint server will not be able to service any page faults for the kernel. As such, the operation does not have to be atomic since no persistent thread which has been marked as copy-on-write will be able to continue until the whole operation has finished. Neither can the TCB of a persistent thread be modified by other threads (e.g., through IPC) while it is marked copy-on-write. Doing so would immediately cause the modifying thread to raise a page fault in the TCB area of the persistent thread, and since the page fault must be served by the checkpointer the modifying thread will also be suspended until the copy-on-write operation has finished. It must be stressed though, that all threads will be able to run while the TCB contents are being written to stable storage. They will only be suspended during the small amount of time it takes to perform the copy-on-write operation.

The checkpointer also contains data structures for keeping track of where the pages in stable storage belong. These data structures must be kept persistent so that the checkpointer can handle page faults correctly after a system restart. Since the checkpointer task itself can not be made persistent (it has to be running in order to take the atomic snapshot), the data structures are written to stable storage as part of the fixpoint.

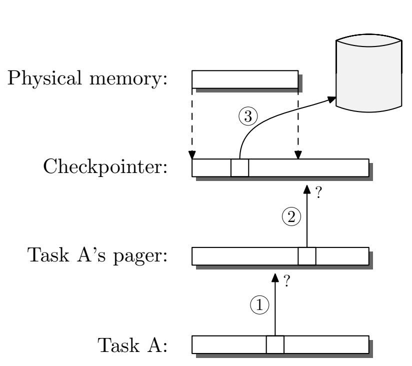
4.2 Checkpointing Kernel State

For a fixpoint to be consistent, no relevant kernel state other than that contained in the TCBs must be required to be persistent. The design of the L4 kernel easily lends itself to this property.

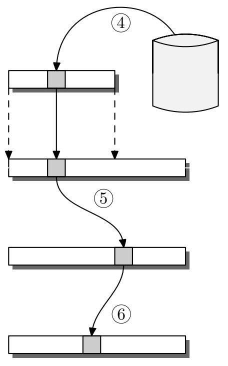
Ready Queues and Waiting Queues. Persistent applications running on top of L4 can not—due to the nature of time sharing systems—determine how fast they will progress through a sequence of machine instructions. In particular, they can not determine how long it will take them to start executing the next IPC system call. As such, they can make no assumptions about the ordering of in-kernel ready queues and IPC waiting queues. The kernel can therefore upon recovery reconstruct the queues in any order it prefers. It just needs to know in which queues the thread should be residing, and this information is stored within the TCB.

Page Tables and Mapping Database. Page tables are an integral part of the μ-kernel data structures. The mapping database is closely related to the page tables. It includes data structures to keep track of sharing between address spaces. Those data structures enable recursive address spaces to be implemented.

Page tables and the mapping database are not included in the data set that is paged to stable storage during a checkpoint. Upon system restart, the mapping database and page tables will therefore all be empty. Once a persistent task is restarted a page fault will immediately be raised. The page fault is caught by the μ-kernel which redirects it to the faulting task's pager. The pager—whose page tables are also empty—will in turn generate another page fault and subsequently cause its own pager to be invoked. Eventually, the page fault will propagate back to the checkpoint server who loads the saved memory contents from stable storage and maps it to the faulting task. Figure 5 illustrates how the page fault is propagated to the checkpointer task, and how the corresponding page frame is mapped recursively into the faulting task.



(a) Page faults are propagated up to the checkpoint server which requests the page frame from stable storage.



(b) Once the page frame is retrieved into memory, mappings are propagated down to the faulting task.

Fig. 5. Cascading page faults. The circled numbers indicate the order in which steps are taken. Arrows in Fig. (a) indicate page fault requests/data transfer requests. Arrows in Fig. (b) indicate data transfers/mappings.

Remaining Kernel State. Other data structures than those mentioned above (e.g., list of available dynamic kernel memory) have no relevance to the state of the kernel as a whole. They are usually only visible in the corresponding kernel functionality (e.g., kernel memory allocator) and are initialized when the kernel is restarted.

4.3 TCB Recovery

Ideally, no threads should be inside the kernel during a fixpoint. However, in reality threads may be blocked due to an ongoing IPC operation, or they may be descheduled during a system call. Strictly speaking, due to the fact that only one thread is allowed to run at a single point in time (the checkpointer), all threads will be inside the kernel during a fixpoint. As such, all system calls must either be restartable, or some other mechanism must exist to put the thread into a consistent state during system recovery.

Fortunately, even though L4 was not designed with persistence in mind, all but one system call in L4 can be safely restarted. When a TCB is recovered and is found to be fixpointed in the middle of one of those system calls, the thread is restarted *in user level* at that location where the system call was invoked so that the same system call will be re-invoked.

Only the IPC system call can not unconditionally be restarted. An IPC operation may consist of two phases, a send phase followed by a receive phase. Separately, both phases are restartable, and, indeed, the IPC operation may very well consist of only one of the phases. However, if a send phase has been completed and a fixpoint is taken before the receive phase has been completed, restarting the entire system call would repeat the send phase a second time after restart although the checkpointed receiver has already received the first message. To avoid this message duplication, IPCs are handled as two-phase operations where the leading send phase is skipped on restart if it has been completed at checkpoint time. The restarted IPC then directly enters its receive phase. Uncompleted sends or single-phase IPCs are entirely restarted.

4.4 Kernel Upgrades

An implication of the fact that almost no kernel state is kept during a fixpoint, is that the μ-kernel may easily be upgraded or modified between system restarts. The essential invariant that must be kept between different kernel versions is that the kernel stack layout of interrupted threads must look the same or that the new kernel version knows the layout of the old version and adapts it to the new layout.

5 Main-Memory Checkpointing

We already have a checkpoint server that acts as a pager for TCBs. This server is easily extended to also act as a pager for all physical memory that is used by implicitly persistent servers and applications. Therefore, the checkpoint server is implemented as a pager located directly below the top level pager. Being almost on top of the pager hierarchy, it automatically has access to all user-level memory of its sibling tasks. In particular, it can unmap page frames temporarily from all other tasks (including all other pagers), map page frames read-only, copy them to buffers, write them to backing store, and remap them afterwards. Thus the checkpoint pager can eagerly or lazily (copy-on-write) checkpoint main memory.

The checkpointing mechanism used is heavily influenced by the mechanisms in EROS [14], KeyKOS [7], and L3 [9]. For each physical page frame, we have allocated two disk blocks, an a-block and a b-block, and a status bit that specifies which of both blocks holds the currently valid checkpointed content of the physical page frame.

At regular intervals, the checkpoint server takes a consistent copy of all threads and address spaces, saving the according physical page frames. Dirty page frames are saved into their a/b-blocks: If the according status bit specifies that the a-block is currently valid the new content is saved into the b-block, otherwise into the a-block. Then, the page frame's status bit is toggled so that the newly written block is marked valid. Non-dirty physical page frames are ignored, i.e., their status bits remain unchanged. Once all modified page frames have been written to disk, the location bit-array itself is atomically written (e.g., using Challis' algorithm [1]), and the checkpoint is completed.

An optimization of the presented algorithm uses write logs to minimize seek time (instead of overwriting the a/b-blocks directly) whenever the amount of dirty page frames is low.

6 Backing-Store Checkpointing

Main-memory checkpointing is not sufficient. In addition, pagers, e.g., anonymous-memory servers, have to be modified so that they support persistence, i.e., implement persistent data objects. Servers that are explicitly built to support persistent objects we call *explicitly persistent* servers.

Many servers are not explicitly built to support persistence. Of course, it would be very helpful to also have a general method that easily extends such servers so that they automatically become *implicitly persistent* servers that synchronize their checkpoints with the checkpoint server.

A method of generating implicitly persistent servers combined with the checkpoint server is sufficient to establish orthogonal persistence. Explicitly persistent servers then specialize and customize the system.

6.1 The Implicit-Persistence Problem of Backing Stores

The checkpoint server only takes care of paging thread control blocks and *physically backed* user-level memory to stable storage. Multiplexing the physical memory among user-level tasks, and swapping parts of virtual memory to stable storage must be taken care of by other pagers. Since those pagers deal with stable storage, writing page frames to disk must be synchronized with the checkpoint server in order to avoid inconsistency. Consider, for example, a scenario in which a checkpoint is made at time c_n, and the next checkpoint is to be made at c_{n+1}. Now, the pager writes a page frame to disk at the time w_1 in between the two checkpoints (i.e., $c_n < w_1 < c_{n+1}$). At a time, w_2, in between the next two checkpoints (i.e., $c_{n+1} < w_2 < c_{n+2}$), the pager then replaces the the on-disk page frame with another page frame. If the machine crashes at a time in between w_2 and c_{n+2}, the system will on recovery be rendered inconsistent because the pager will believe that the on-disk page frame is the one that it wrote on time w_1. It has no idea that it changed the contents at time w_2.

For implicit persistence, we need a method to make any pager (or server) that uses the stable backing store persistent; without modifying or extending it. This is achieved by the recoverable-disk driver.

6.2 The Recoverable-Disk Driver

The *recoverable disk* is a user-level server (driver) implementing a logical disk that offers the operations *checkpoint* and *recover* besides normal *read-block* and *write-block*. Disk writes remain fragile until they are committed by a subsequent *checkpoint* operation. The *recover* operation sets the entire disk back to the status of the last successful *checkpoint* operation, i.e. undos all writes after the last completed checkpoint.

All implicitly persistent servers use the recoverable-disk driver instead of the raw disk driver for reading and writing blocks. Furthermore, the checkpoint server synchronizes its main-memory checkpoints with checkpoints in the recoverable-disk driver. As a consequence, all pagers using both memory from the checkpointer and disk blocks from the recoverable-disk become implicitly persistent.

The recoverable-disk implementation uses methods similar to shadow paging [12] and to the algorithms used for main-memory checkpointing (see Sect. 5). It offers n logical blocks; however, physically, it uses $2n$ blocks. Each logical block k is mapped either to physical block a_k or b_k without overlaps.[3] A bitmap *CurrentBlock* specifies for each k whether the a-block or the b-block is currently valid. For a 4 GB recoverable disk and 4 KB blocks, this bitmap requires 128 KB of memory.

A second bitmap of the same size, *CurrentlyWritten*, is used to determine which logical blocks have been written in the current checkpoint interval: Assume that a server/pager writes to logical block k, and a_k is the currently associated physical block. If $\mathit{CurrentlyWritten}_k$ is not set, then b_k becomes associated to k, and the according *CurrentlyWritten* bit is set. The physical block is then written. If $\mathit{CurrentlyWritten}_k$ is already set when writing the block, the *CurrentBlock* and *CurrentlyWritten* bitmaps stay as-is, and the block is simply overwritten.

When a checkpoint occurs, the *CurrentBlock* bitmap is atomically saved on stable storage and all *CurrentlyWritten* bits are reset to zero.

Modifications of this algorithm permit multiple valid checkpoints. Furthermore, the number of totally required physical blocks can be reduced to $n + m$ if at maximum m blocks can be written between two checkpoints.

The recoverable-disk driver enables legacy pagers (or other servers accessing disk) to be used in the persistent system without modification. If a pager for efficiency reasons wants to bypass the recoverable-disk driver, it can of course do so. Such pagers, however, should synchronize with the checkpoint server in order to keep the system in a consistent state.

6.3 Device Drivers and File Systems

Device Drivers. As mentioned before, most device drivers can not be made persistent. Care must therefore be taken to not include any state about other threads in the system within the driver. For example, a network interface driver should

[3] $a_k \neq b_k$ for all k, and $a_k = a_{k'} \lor b_k = b_{k'} \Rightarrow k = k'$.

not include any knowledge about open network connections since this knowledge would be lost during a system reboot. In general, however, this restriction on device drivers does not impose any problems. In the case of the network interface driver for instance, the knowledge of open connections would reside in some network protocol server—a server that would be included in the set of persistent tasks.

File Systems. Having a fully transparent persistent system somewhat obviates the need to support an explicitly persistent file system. There are some cases, however, that point in the direction of having a file system: First of all, many users are simply used to dealing with files and file hierarchies. There is no need to upset users unnecessarily. Moreover, there might be a need to interchange data with other systems. Files are a nice and clean way to handle interchangeability on widely different platforms. Most important though, is that many applications today rely on being backed by some sort of file system. Portability of applications is therefore a key issue.

Since tasks are fully persistent, a UNIX-like file system can easily be implemented as a collection of servers (as in L3 [9] or EROS [14]). In short, a directory or a file is nothing more than a persistent object. As such, a persistent server is used to represent each directory and each file. When another task in the system accesses a file, the requests will be directed to the corresponding file server (or directory server).

Of course, a file system server may also be implemented as a traditional file system explicitly accessing stable storage. If this is the case, they will have to use the recoverable-disk driver.

7 Related Work

The concept of transparent orthogonal persistence is not new. In fact, system-wide persistence was an integral part of L4's predecessor—the L3 μ-kernel [9]. With L4, however, persistence is not an integral part of the kernel. It is an add-on feature that the user may choose to ignore.

Other μ-kernels integrating persistence into the kernel include EROS [14] and its predecessor KeyKOS [7]. With EROS, taking a snapshot of the system also includes performing a consistency check of critical kernel data structures. This catches possible kernel implementation bugs, and prohibits these bugs to stabilize in the system. A checkpoint in L4, on the other hand, does not include any critical kernel data structures. Doing such a consistency check is therefore unnecessary.

In contrast with L3 and EROS, Fluke [15] is a μ-kernel offering transparent checkpointing at user-level. This is achieved by having the kernel export user-visible, partly pickled, kernel objects. The checkpointer pickles the remaining parts of the objects and saves them to stable storage together with the memory-images of the tasks.

Most transparently persistent operating systems are based upon μ-kernels. An exception to this rule is Grasshopper [2]. Grasshopper hopes to achieve persistence through the use of some specially designed kernel abstractions. Based on the experiences learned from Grasshopper, the designers have later created a μ-kernel based operating system, Charm [3], aiming at supporting persistent applications. With Charm, no particular persistence model is enforced upon the applications. The kernel instead provides the application system with mechanisms to construct their own persistence policy. In short, all in-kernel meta-data (such as page tables) are exposed to the application, and the application is itself responsible for making this data persistent.

Several other facilities for user-level checkpointing have been implemented [4, 8, 11, 13]. Typically, the `fork()` UNIX system call is used to periodically create a snapshot of the task's image, allowing the application to continue execution while the image is being written to stable storage. Such systems, however, can not manage to restore much of the operating system state upon recovery. Neither can they assure that their interaction with the surrounding system will leave the checkpoint in a consistent state. As such, these checkpointing facilities are only usable within a certain type of scientific applications.

8 Conclusions

We have seen that user-level transparent orthogonal checkpointing can readily be implemented in the context of a second-generation L4 μ-kernel. The checkpointing facility relies for the most part on existing kernel abstractions. Only a minimal set of additions to the μ-kernel is needed. The key for implementing persistence on top of the μ-kernel is its concept of recursive address spaces that enables implementation of all main-memory management by user-level servers.

Consequently, orthogonal persistence can be implemented almost entirely through user-level pagers and drivers, particularly through a checkpoint server and a recoverable disk driver. Both are user-level servers and do not depend on the OS personality that runs on top of the μ-kernel. The resulting support of orthogonal persistence is thus widely transparent to and independent from application *and OS*.

The transparent checkpointing design presented here is currently in the process of being implemented. The system will be used in conjunction with L^{4}Linux [6] as well as with the multi-server SawMill system [5] on top of L4.

References

1. Michael F. Challis. Database consistency and integrity in a multi-user environment. In *Proceedings of the 1st International Conference on Data and Knowledge Bases*, pages 245–270, Haifa, Israel, August 2–3 1978. Academic Press.
2. Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: an orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994.

3. Alan Dearle and David Hulse. Operating system support for persistent systems: past, present and future. *Software – Practice and Experience, Special Issue on Persistent Object Systems*, 30(4):295–324, 2000.
4. Elmootazbellah N. Elnohazy, David B. Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Houston, TX, October 5–7 1992.
5. Alain Gefflaut et al. Building efficient and robust multiserver systems: the SawMill approach. In *Proceedings of the ACM SIGOPS European Workshop 2000*, Kolding, Denmark, September17–20 2000.
6. Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastiann Scönberg, and Jean Wolter. The performance of μ-kernel bases systems. In *Proceeding of the 16th ACM Symposium on Operating System Principles (SOSP)*, Saint-Malo, France, October 5–8 1997.
7. Charles R. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the 2nd International Workshop on Persistent Object Systmes (POS2)*, Paris, France, September 24–25 1992.
8. Juan León, Allan L. Fisher, and Peter Steenkist. Fail-safe PVM: a portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.
9. Jochen Liedtke. A persistent system in real use: experiences of the first 13 years. In *Proceedings of the 3rd International Workshop on Object-Orientation in Operatins Systems (IWOOOS '93)*, Asheville, NC, December 9–10 1993.
10. Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, Copper Mountain Resort, CO, December 3–6 1995.
11. Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report #1346, University of Wisconsin-Madison, April 1997.
12. Raymond A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems (TODS)*, 2(1):91–104, September 1977.
13. James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: transparent checkpointing under UNIX. In *Proceeding of the USENIX 1995 Technical Conference*, New Orleans, LA, January 16–20 1995.
14. Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Pronciples (SOSP '99)*, Kiawah Island Resort, SC, December 12–15 1999.
15. Patrick Tullmann, Jay Lepreau, Bryan Ford, and Mike Hibler. User-level checkpointing through exportable kernel state. In *Proceedings of the 5th International Workshop on Object-Orientation in Operating System (IWOOOS '96)*, Seattle, WA, October 27–28 1996.