Supporting Persistent Object Systems in a Single Address Space[1]

Kevin Elphinstone, Stephen Russell, Gernot Heiser[2]
School of Computer Science & Engineering,
The University of New South Wales, Sydney 2052, Australia

Jochen Liedtke
GMD SET-RS, Schloß Birlinghoven, 53757 Sankt Augustin, Germany

UNSW-CSE-TR-9601 — 28 February 1996

[1] This work was supported by grants from the Australian Research Council (ARC) and the German Ministry for Research and Technology (BMFT).

[2] E-mail: G.Heiser@unsw.edu.au, fax: +61 2 395 5995, phone: +61 2 395 5156

Abstract

Single address space systems (SASOS) provide a programming model that is well suited to supporting persistent object systems. In this paper we show that stability can be implemented in the Mungi SASOS without incurring overhead in excess of the inherent cost of shadow-paging. Our approach is based on the introduction of aliasing into the SASOS model and makes heavy use of user-level page fault handlers to allow implementation outside the kernel. We also show how the demands of database systems for control over page residency and physical I/O can be accommodated. An approach to user-level implementation of distributed shared memory (DSM) coherency models is outlined.

1 Introduction

Single address space operating systems (SASOS) such as Angel [1], Opal [2] and Mungi [3] are based on the idea that a single, large virtual address space holds all data in a (potentially distributed) computing system. It has been pointed out before [4] that this class of operating systems provides a natural solution to an old problem of persistent programming: How to save arbitrary data structures on secondary storage having to translating them first into a form that allows reconstruction in memory at a later time. In a SASOS this becomes a non-problem: Secondary store is solely a backing device for virtual memory (VM) paging, and data are guaranteed to always appear at the VM location at which they were originally created. Hence internal references continue to work without any need for translation.

The single-level store of a SASOS is, in principle, persistent in the sense that data allocated in memory can outlive the process that created them, and also survive an orderly system shutdown and restart. However, stability in case of an unplanned system shutdown, e.g. in the case of a power failure, is a much harder problem. Classical approaches to achieving stability are logging and shadow paging. It is obviously possible to implement such stability schemes in the kernel, as was done in Monads [5]. However, this dictates a specific stability model to applications, and makes it difficult to adapt to specific needs of applications. For example, some applications never require stability, and should not have to pay the runtime overhead, while other applications require stability at certain times, others again require it constantly. It is, therefore, desirable to give the application maximum control over the stability model. It is not, *a priori*, clear that this can be done in a SASOS without incurring overhead in the form of having to keep redundant memory copies or to perform extra copy operations.

Object-oriented database systems, such as ObjectStore [6] or O_2 [7], seem well-suited for implementation on top of a SASOS, as their data model fits the SASOS model very well. In fact, one of the main difficulties facing the implementation of these system is the need to construct and efficiently handle system-wide unique object identifiers. A SASOS provides these for free in the form of VM addresses. However, database management systems (DBMS) tend to be very demanding customers for an operating system: For efficiency reasons they need to control residency of data in memory, and they need control over backing store allocation in order to reduce disk seek times. Can such systems be accommodated in a SASOS, which is characterised by a single-level store and hence the absence of an I/O model, without breaking the SASOS paradigm?

We will address these issues in the remainder of this paper. We will demonstrate that by introducing aliasing into the SASOS model we can, with the help of user-level pagers, implement shadow-paging efficiently in the Mungi system. We will also show that DBMS demands for control of physical memory (PM) and physical I/O can be met, without creating a need to integrate the DBMS, or parts of it, into the kernel.

2 Support for Persistent Object Systems in Mungi

2.1 Mungi Overview

In Mungi, virtual memory is allocated in contiguous, page-aligned segments called *objects*. Objects are also the unit of protection: Access rights are object-based. The system makes no assumptions about the internal structure of objects.

Access control is based on password capabilities, i.e. (base address, password) pairs. The access rights conferred to a holder by a capability are a combination of read, write, execute and delete. A capability conferring all these rights is called an *owner capability*. A system wide directory called the *object table* contains information about all objects, including their set of valid capabilities and corresponding access rights. Capabilities can be added or revoked by a holder of an owner capability.

When an object is first accessed by a thread, a page fault will occur as there exists no virtual to physical memory (VM→PM) mapping for the thread. The kernel will validate the access by matching the thread's list of capabilities with the set recorded for that object in the object table. If the access is valid, the kernel caches the validation information and invokes the appropriate page fault handler to set up a mapping for the accessed page.

In the following subsections we will present the extensions proposed to Mungi for supporting persistent object systems. These are all concerned with the relationship between physical and virtual memory. They are summarised in Fig. 1 and explained in the remainder of this section.

InstallPager (Capability obj, Capability pager);
Flush (Address adr, Int length)
Copy (Address from, Address to, Int length) Map (Address from, Address to, Int length; Mode m) Unmap (Address to, Int length)

Figure 1: Mungi system calls dealing with the relationship between virtual and physical memory.

2.2 User-level pagers

In Mungi, each object is associated with a *pager* which will be called by the system to handle page faults. When an object is first allocated, its pager is the system's default pager, which performs normal VM paging. A user-level pager (ULP) can be installed with the InstallPager system call. A null pager capability will re-install the default pager. Owner capability is required for changing an object's pager.

Pagers are invoked in three cases: access of a non-resident page (*residency fault*), write attempt to a read-only page (*write fault*), and a Flush call executed on a page (*flush event*).

The default pager handles a write fault by signalling a protection violation to the offending thread. Default handling of a residency fault results in allocating a physical frame to it, which is either zero-filled if the virtual page has not been allocated before, or reloaded from disk. Default pager action on a flush event is explained below.

2.3 Flushing dirty pages

The Flush system call causes invocation of the appropriate pager with a flush event. Both adr and length must be page-aligned, and all affected pages must be part of the same object.

The default pager handles flush events by writing the pages' contents to disk if they are dirty **and** ensures that the kernel's mappings of these pages to backing store are flushed as well. The kernel uses a stable logging scheme for storing these mappings on disk. No write will occur on a clean page.

2.4 Virtual memory mapping primitives

The mapping primitives Map and Unmap are based on similar calls in the L3 operating system [8] (although semantics differ significantly from the L3 operations). Copy is related to these as it also affects mappings. Address and length parameters must be page aligned, and if a range of pages is specified, all pages within the range must be part of the same object. The mode can either be read-only or read/write.

2.4.1 Copy-on-write

The Copy system call performs page-wise memory copy. Page faults will be generated if the source or destination pages are not resident, or if the destination pages are R/O. The source and destination ranges must not overlap. Read capability on the source and R/W capability on the destination is required.

If the destination is handled by the default pager, the implementation uses copy-on-write. This not only avoids copying pages which never get modified, but also avoids zeroing a newly allocated page immediately before overwriting it. If the target of a Copy is flushed or aliased at a later time, the physical copy will be forced if it has not already been made. If the source is later unmapped, the destination inherits the source's physical mappings (including backing store) if the physical copy has not yet been made. Mapping of the source pages to backing store will not be changed by a Copy operation.

2.4.2 Map

Map(from,to,length,mode) creates an alias in the virtual address-space by mapping VM addresses [to,to+length-1] to the same physical frames as VM addresses [from,from+length-1]. If the destination was aliased to another page, the alias will vanish.

The two address ranges must not overlap, unless they are identical, in which case the call serves to modify the access mode (R/O or writable) without changing any VM→PM mappings. This call is a no-op on non-resident pages (i.e. will not fault them into PM).

R/W capability is required on the objects affected by a Map, even in the case of a R/O mapping. The operation will cause a page fault on the source pages if they are not resident. Fig. 2 shows the effect of a Map operation.

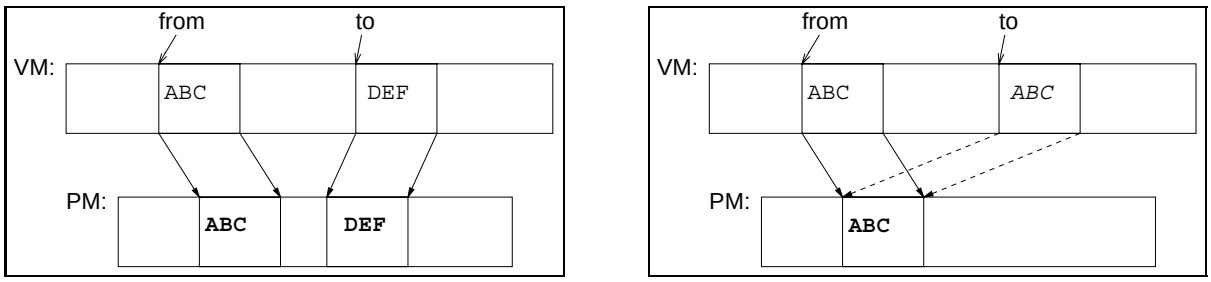

Figure 2: Address-space before (left) and after (right) execution of a Map(from,to,length,ro) system call. Dashed arrows indicate R/O mappings, slanted text indicates R/O data.

An alias established by a Map operation only exists as long as the pages are resident. This implies that the Map operation is essentially only of use to ULPs, as it needs to be reestablished on any residency fault. For the same reason, a Map operation whose destination pages are handled by the default pager makes no sense and is therefore invalid.

2.4.3 Unmap

The aliasing in Fig. 2 can be undone with the operation Unmap(to,length); however, Unmap can also be applied to pages which are not aliased. On return, to does not have any mapping to PM or backing store, as if newly allocated. Hence, if a page to be unmapped was an alias, the alias is simply removed; if another page was mapped to it, both pages will become unmapped. Unmapping also removes any write protection from the specified address range. Unmapping a page which is neither resident nor backed on disk is a no-op. R/W capability is required for performing an Unmap operation.

For an object serviced by the default pager unmapping is semantically equivalent to zeroing all memory at addresses [to,to+length-1], as Mungi's default pager uses zero-on-read for uninitialised memory. Fig. 3 shows an example of an Unmap operation.

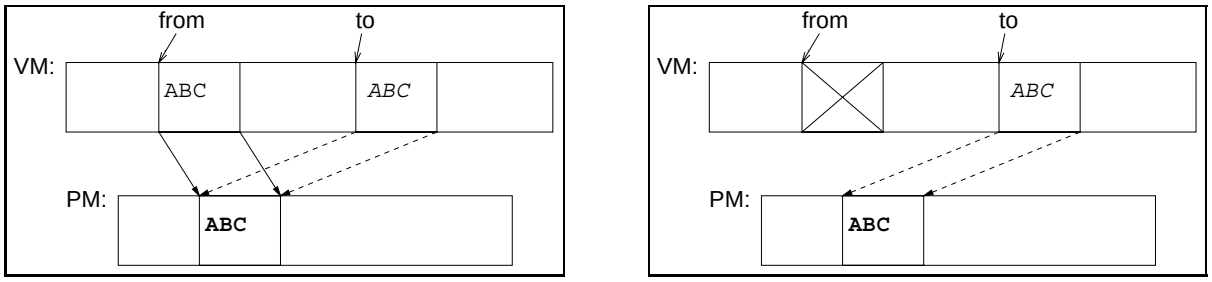

Figure 3: Address-space before (left) and after (right) execution of an Unmap(from,length) system call.

2.5 Recursive address spaces?

Systems like Grasshopper [9] and L4 [10] use similar mapping operations, but use them in a hierarchical fashion to recursively construct address spaces. This does not fit the SASOS model, as there can only be one address-space, so mappings can only operate between different parts of the same address-space. However, pagers can be nested in our model: If a ULP handles a page fault by mapping another page onto the faulting one, this may trigger a fault on the source page, invoking its ULP. This will continue until a resident page is mapped to the faulting one, or the default pager is invoked.

3 Implementation of Shadow-Paging

If we ignore stability issues we can checkpoint an object simply by copying it. To roll back, the checkpoint is copied back onto the object. The copy-on-write semantics of the default pager for the Copy system call ensures that only dirty pages are actually copied.

A stable version of the checkpointing scheme can be built with the help of a user-level pager using explicit shadow-paging [11] and an atomic update operation [12]. For the purpose of using Challis' atomic update algorithm we use a two-page object st_page which is located at a well-known address (e.g. pointed to by the checkpointed object's entry in the object table). We can then do atomic updates to stored values as shown in Fig. 4. The basic idea of the algorithm is to alternate between two pages for writing data to disk, so that in case of a system failure during a write operation there is always a stable copy left. Pages are written with a timestamp at the beginning and the end. When reading them back, the timestamps are compared and a page is considered stable if its timestamps agree. If both pages are stable, the timestamp identifies the most recent copy.

To implement shadow paging for object O_0, we allocate two additional objects, O_1 and O_2, of the same size as O_0. O_1 and O_2 are both handled by the default pager. Note that the allocation of an object only allocates virtual memory. No physical memory is allocated until the object is actually accessed.

The addresses of O_1 and O_2 are held in the stable object st_page, as is a bitmap which indicates for each page of O_0 to which of the two shadow objects it is mapped.

Fig. 5 shows the algorithm (ignoring locking). The variable st_index indicates which page of st_page holds the bitmap for the stable copy of O_0. The variable map holds the bitmap of the dirty copy of O_0, while dirty identifies pages which have been modified since the last checkpoint. WriteProtect has been used as an alias for a Map call which only serves to make an object read-only, and size parameters have

```
typedef struct St_map {
  Address a[2];
  Bool pm[...];
}
typedef struct St_page {
  Date d_0;
  St_map m;
  Char fill[...];
  Date d_1;
}
St_page st_page[2];

put_stable (Int index, St_map m) {
  st_page[index].d_0 = timestamp();
  st_page[index].m = m;
  st_page[index].d_1 = st_page[index].d_0;
  Flush(st_page[index],ps);
}
```

```
get_stable (Int &index, St_map &m) {
  if (st_page[0].d_0 == st_page[0].d_1)
    if (st_page[1].d_0 == st_page[1].d_1)
      if (st_page[0].d_0 > st_page[1].d_0)
        index = 0;
      else
        index = 1;
    else
      index = 0;
  else
    index = 1;
  m = st_page[index].m;
}
```

Figure 4: Pseudocode implementation of Challis' algorithm.

been omitted for clarity. The procedures current and other query the page map while swap_current_other modifies it.

The Copy, Unmap and Flush system calls in the setup code serve to dissociate the physical backing store from O_0 and associate it with O_1. The copy-on-write semantics of the Copy operation ensure that this happens without actual copying. This sequence is not required if stable checkpointing is used from the time O_0 is allocated.

After setup, O_1 holds the stable copy of the object. On a write fault a new page is allocated and the corresponding bit in map is flipped to indicate the location of the dirty page, as shown in Fig. 6. On a checkpoint, all dirty pages are flushed and the present state recoded as the stable one. After the first checkpoint, some stable pages belong to O_1, others to O_2, depending on how often they have been modified.

While the mappings of VM pages to backing store for the backup pages referenced by O_1 and O_2 need to be persistent (ensured by Flush calls in Fig. 5), this is not the case for the mappings of O_0 or those of dirty pages in O_1 and O_2. As a consequence, no mappings need to be made persistent between checkpoints, hence no Flush calls need to be performed by the pager.

Objects O_1 and O_2 are hidden from the application; all data accesses by application programs go through O_0, thereby ensuring that all embedded references work correctly. The algorithm can easily be applied to consistently checkpoint a set of related objects, the stable object st_page then contains object references and bitmaps for all the involved objects.

4 Other Applications of the Model

4.1 Controlling page residency

Certain applications, like DBMS, require the ability to lock pages in memory for efficiency reasons. To support this, we create a special region, σ_0, in virtual memory to map all of PM. This region is "magical", i.e. the kernel knows about its special relationship to PM. Any page in VM is resident *if and only if* it is

```
St_map map;
Bool st_index;
Bool dirty[...];
```

```
setup() {
  map = { {O1, O2}, 0};
  dirty = 0;
  st_index = 0;
  Copy(O0, O1); /* copy-on-write */
  Unmap(O0); /* leave mapping to pager */
  Flush(O1); /* make sure backup is stable */
  InstallPager(O0, pager);
  put_stable(st_index, map);
  put_stable(!st_index, map);
}
```

```
checkpoint() {
  WriteProtect(O0);
  Flush (O1); /* only flushes dirty pages */
  Flush (O2);
  st_index = !st_index;
  put_stable(st_index, map);
  dirty = 0;
  for (all pages p)
      /* unmap old backups: */
    Unmap(other (map, p));
}
```

```
pager(faulting_page) {
  current_page = current (map, faulting_page);
  other_page = other (map, faulting_page);
  if (write_fault)
      /* create dirty page: */
    Copy(current_page, other_page);
    Map(other_page, faulting_page, rw);
    swap_current_other(map);
      /* no Flush needed! */
    dirty[faulting_page] = TRUE;
  elsif (not_resident) /* re-establish mapping: */
    if (dirty[faulting_page])
      Map(current_page, faulting_page, read_write);
    else
      Map(current_page, faulting_page, read_only);
    /* ignore flushes */
}
```

```
rollback() {
  get_stable(st_index, map);
  for (all pages p)
      /* free dirty blocks: */
    Unmap(other (map, p));
  dirty = 0;
}
```

Figure 5: Stable checkpointing of object O_0.

aliased to a page in σ_0. No page fault will ever occur in σ_0; is is essentially a permuted frame table.

The system administrator has the option of allocating an object O_σ in σ_0. Such an object is known to the kernel to be unpagable. By handing a R/W capability for O_σ to a DBMS, the system administrator thereby enables the DBMS to lock pages in PM by aliasing them to O_σ. In the example in Fig. 7, page q has been locked by a

Map(q, O_σ.page[n])

system call. The page can be unlocked and the locking slot reused to lock page p by executing

Map(p, O_σ.page[n]).

Simply doing

Unmap(O_σ.page[n])

unlocks the page without locking another one in its place.

The advantage of this scheme over introducing special system calls to lock and unlock pages is that it ties in neatly with our capability-based protection scheme. By handing out a capability to an object of a particular size, the system allows the holder of the capability to lock a strictly limited number of pages.

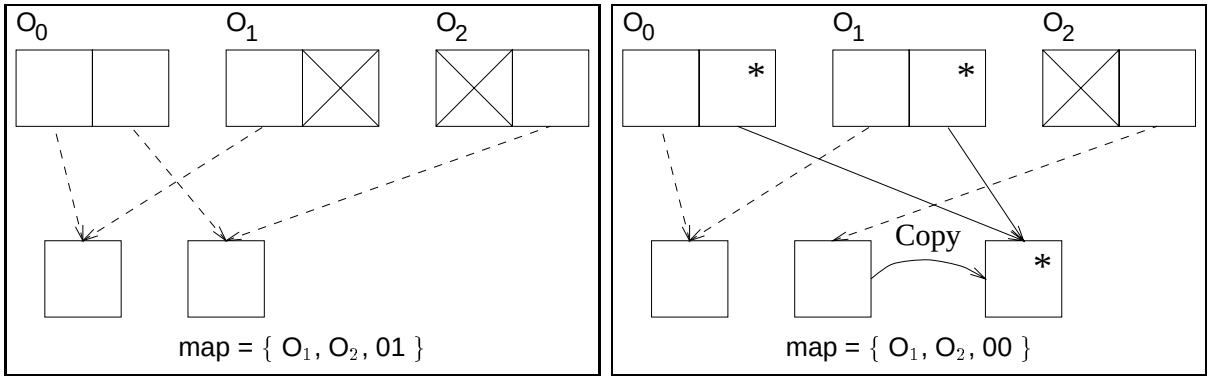

Figure 6: Copy-on-write operation in the stable checkpointing example. Mappings are shown at the time the fault is taken (left) and when the pager returns (right). An asterisk indicates a dirty page, crossed-out pages are unmapped. The current page map is shown at the bottom.

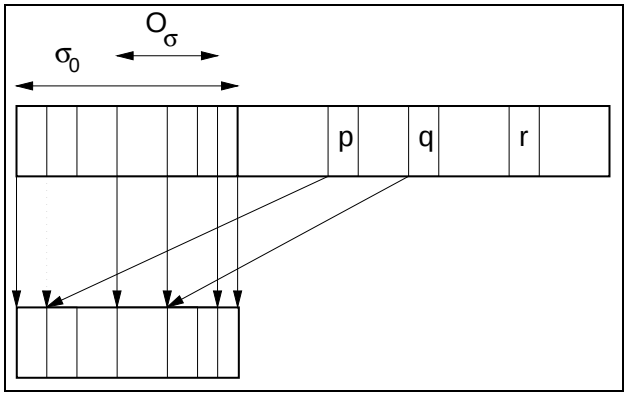

Figure 7: Locking pages in PM by aliasing to O_σ: Page p is resident and pageable, page q is locked in memory (unpagable), page r is not resident. Page p's alias in σ_0 is only known to the kernel.

4.2 Physical Disk I/O

As the SASOS model presents the user with a single-level store, I/O is not part of the model but is hidden in the operating system. This has significant advantages in simplifying application code. However, there are cases where applications (like DBMS) require control over I/O operations for efficiency.

A certain amount of control over I/O is already available through ULPs and the Flush system call. However, in order to optimise seek times, some applications may want to have control over the actual allocation of pages on the disk.

We can support this in Mungi by mapping all of secondary storage into the virtual address space, as another special region σ_d. An object O_d is allocated covering all (or part) of σ_d, and R/W capability to that object is given to the application. The DBMS can then map virtual pages to specific disk blocks by mapping the appropriate pages of O_d to its memory objects.

To combine I/O control with control over page residency, the application needs to alias some pages of O_d twice—once to the data object and once to σ_d. To lock page p of object O, backed by disk block q, into physical frame r, the DBMS needs to execute

Map (O_d.page[q], O.page[p], ...);
Map (O_d.page[q], O_σ.page[r], ...).

To enable an application to optimise its physical I/O, it needs to be given information on the physical layout of the disk (number of tracks, cylinders, etc.) This information can be stored in a standard format at a well-known location, e.g. the first disk block.

Note that, in spite of the large amount of disk storage mapped into VM, this approach does not use up a significant fraction of VM: Mungi's total virtual address space is 2^{64} bytes or sixteen billion gigabytes.

4.3 Network I/O and distributed shared memory

Our last example is not directly dealing with persistence but is included because it deals with distribution, which is certainly of great practical importance to persistent systems. We show how a distributed shared memory (DSM) coherency protocol can be implemented *at user level* in Mungi.

As above we map all of PM into VM, each node's PM is mapped to VM at a different location. Node n_i's memory is mapped to VM region σ_0^i, and the kernel on n_i ensures that no $\sigma_0^j, i \neq j$ is ever accessed by n_i. The kernel also needs to ensure that no thread with access to σ_0^i ever migrates.

Assume a thread executing on node n_0 triggers a residency fault on page p. When the appropriate pager is called it requests the page from the network (via broadcast or other appropriate means). Assume the page happens to be resident on n_1, i.e. it has been mapped into σ_0^1. Node n_1 sends the contents of the page to n_0. Depending on the coherency model used, the pager on n_1 may first unmap or write-protect the page and its aliases (see Fig. 8).

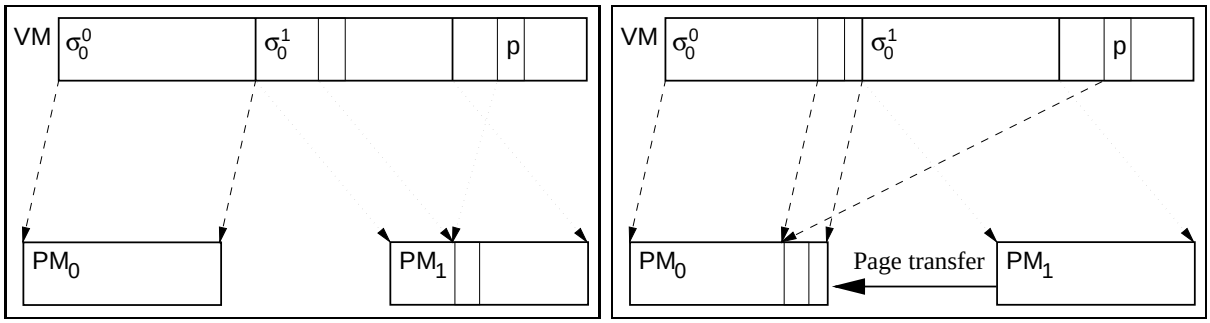

Figure 8: DSM paging: Page p is originally resident on node n_1 (left) but transferred to node n_0 (right). Broken arrows indicate VM→PM mappings of node n_0, while dotted arrows represent mappings of node n_1.

The default pager implements a default DSM protocol (e.g. multiple reader, single writer). This takes care of such cases as the user level DSM pager not presently residing in the node's memory or backing store.

Obviously, we have only sketched the DSM implementation, glossing over most of the details. In particular, a complete solution will need to integrate disk I/O with network I/O.

5 Conclusions

In this paper we have shown that stability based on shadow paging can be elegantly integrated into a SASOS, without the need to implement it in the kernel. We have demonstrated that shadow-paging can be implemented at the user level if we introduce virtual memory aliasing and use user-level page fault handlers. This can be achieved without performance overhead in the form of extra copying operations or redundant copies being kept on disk or in RAM.

We have shown that the same mechanisms can be used to support database systems by allowing them to lock pages in RAM and to control physical I/O. A user-level implementation of DSM coherency models was also outlined.

The question arises whether the resulting system is still a proper SASOS. The answer is "yes", as the basic SASOS paradigm is still valid: each user thread sees the same[1] data at a given address (if they see anything at all); there is still just a single address space. Furthermore, our mapping operations do not allow users to do anything they could not do by copying, but they ensure that it can be done efficiently.

We have not addressed stability issues with respect to distribution. However, having shown that we can employ the usual approach to solve the problem of stability on a single node, there is no reason to believe that schemes providing stability in other distributed systems cannot be used in ours.

References

[1] K. Murray, A. Saulsbury, T. Stiemerling, T. Wilkinson, P. Kelly, and P. Osmon. Design and implementation of an object-orientated 64-bit single address space microkernel. In *Proceedings of the 2nd USENIX Symposium on Microkernels and other Kernel Architectures*, pages 31–43, September 1993.

[2] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12:271–307, November 1994.

[3] J. Vochteloo, S. Russell, and G. Heiser. Capability-based protection in the Mungi operating system. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, pages 108–15, Asheville, NC, USA, December 1993. IEEE.

[4] S. Russell, A. Skea, K. Elphinstone, G. Heiser, K. Burston, I. Gorton, and G. Hellestrand. Distribution + persistence = global virtual memory. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems*, pages 96–99, Dourdan, France, September 1992. IEEE.

[5] F. A. Henskens and J. Rosenberg. Distributed persistent stores. *Microprocessors and Microsystems*, 17:147–59, 1993.

[6] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):51–63, October 1991.

[7] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[8] J. Liedtke. A persistent system in real use—experience of the first 13 years. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, pages 2–11, Asheville, NC, USA, December 1993. IEEE.

[9] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, and F. Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, 1994.

[10] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, Copper Mountain, CO, USA, December 1995.

[11] R. A. Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems*, 2:91–104, 1997.

[1] In a distributed system, the word "same" needs to be taken with a grain of salt, as some threads may see a somewhat newer copy than others. This is not different from any other distributed shared memory system.

[12] M. F. Challis. Database consistency and integrity in a multi-user environment. In B. Shneiderman, editor, *Databases: Improving Usability and Responsiveness*, pages 245–70. Academic Press, 1978.