Implementation and Performance of the Mungi Single-Address-Space Operating System

Gernot Heiser Kevin Elphinstone Jerry Vochteloo Stephen Russell

Department of Computer Systems
School of Computer Science and Engineering
The University of New South Wales, Sydney 2052, Australia
E-mail: {gernot,kevine,jerry,smr}@cse.unsw.edu.au
WWW: http://www.cse.unsw.edu.au/~disy

Jochen Liedtke

IBM T. J. Watson Research Center
30 Saw Mill River Road, Hawthorne, NY 10532, USA
E-mail: jochen@watson.ibm.com

UNSW-CSE-TR-9704 — June 1997

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Single-address-space operating systems (SASOS) are an attractive model for making the best use of the wide address space provided by the latest generations of microprocessors. SASOS remove the address space borders which make data sharing between processes difficult and expensive in traditional operating systems. This offers the potential of significant performance advantages for applications where sharing is important, such as object-oriented databases or persistent programming systems.

Previously published SASOS were not able to demonstrate these performance advantages. We have built the Mungi system to show that these advantages can indeed be realized. Mungi is a very "pure" SASOS, featuring an unintrusive protection model based on sparse capabilities, a fast protected procedure call mechanism, and uses virtual memory as the exclusive inter-process communication mechanism, as well as for I/O. We believe this simplicity of our model makes it easy to implement it efficiently on conventional architectures.

Our realization of Mungi for the MIPS R4600 64-bit microprocessor is presented, which is based on our implementation of the L4 microkernel. Mungi is shown to outperform a well-tuned commercial operating system in several important aspects, such as task creation and inter-process communications, and on the OO1 object-oriented database benchmark. This demonstrates clearly that the SASOS concept is viable, and that a well-designed microkernel is an excellent base on which to build high-performance operating systems.

1 Introduction

Single-address-space operating systems (SASOS) have recently been proposed as an attractive model for making the best use of the wide address space provided by the latest generations of microprocessors [WSO+92, CLLB92, RSE+92]. The basic idea of these systems is that by removing address space boundaries, they encourage sharing of data between processes.

In a SASOS, all processes share the same address space. This address space is decoupled from the lifetime of any process, and all objects created within it are potentially persistent, eliminating the need for a file system. As all data live in the shared address space, and are seen at the same virtual address by all processes, pointers have the same meaning for each process. Arbitrary data structures can then be shared without the need for techniques such as marshaling or pointer swizzling.

While this simplified programming model makes many programming tasks much easier, the major application areas likely to benefit most from this approach are those that require efficient support for persistence. Examples of these include object-oriented database systems and persistent programming languages.

Besides these advantages for application programmers, there are also benefits on the system level. A SASOS avoids problems with virtual caches created by address aliasing in multi-address-space systems: as every datum is always accessed through the same address, different cache lines are guaranteed to refer to different data. It has also been claimed [WM96] that the simplified model significantly reduces the complexity of the operating system, and leads to improved performance.

A number of SASOS prototypes have been implemented to date, for example, Opal [CLFL94] and Angel [WM96]. These implementations were intended primarily as a proof-of-concept and have not been able to fully demonstrate the potential advantages of a SASOS. In this paper we present implementation details and some performance figures for the Mungi operating system, which we believe is the first native implementation of a SASOS on standard 64-bit hardware.

In designing Mungi, we decided to build a SASOS which was as pure as possible, without sacrificing support for features that we deemed essential, such as protection, encapsulation, and orthogonal persistence. We decided to take the memory-only model as far as possible, and eliminated explicit support for I/O and conventional inter-process communication (IPC). Our measurements show that this approach works: The system can outperform, by a significant margin, traditional systems if applications make full use of our model.

The paper is structured as follows: Section 2 presents an overview of the Mungi system, detailing the basic abstractions of our model. For this model to be accepted as effective, it is also necessary to demonstrate that these abstractions can be implemented efficiently. Section 3 describes how we have built the Mungi model on top of our implementation of the L4 microkernel [Lie96a] for the MIPS R4600 processor. In Section 4 we review related work on capability systems and other global address space approaches, as well as recent implementations of SASOS. Benchmark results for task creation and deletion, cross-domain calls, and object-oriented database operations, show performance significantly exceeding that of a commercial operating system. These are presented in Section 5, followed by conclusions in Section 6. Appendix A describes the simplifications we made to the OO1 benchmark to suit the aims of this paper.

2 Overview of the Mungi System

The basic abstractions provided by Mungi are: *capability*, *object*, *task*, *thread*, and *protection domain*.

Objects are the basic storage abstraction. They consist of a contiguous range of pages, with no further structure imposed by the system. Objects are protected by capabilities which are described below.

Threads are the basic execution abstraction. A task is a set of threads which share a protection domain. A protection domain consists of a set of capabilities. Capabilities are presented implicitly by storing them in a special data-structure known to the system. This reduces the need for most applications to deal with capabilities and thus makes protection transparent.

There are no explicit system calls to support I/O in Mungi. Instead, I/O devices are mapped into virtual memory, and user-level page fault handlers and virtual memory mapping operations are used for dealing with these devices.

The remainder of this section describes in more detail the basic Mungi abstractions. A full description of the API can be found under the Mungi WWW pages from URL http://www.cse.unsw.edu.au/~disy/Mungi.html.

2.1 Capabilities

Capabilities confer to their holders rights to perform specific operations on objects. When an object is created, an *owner capability* to that object is returned, giving the holder full rights to the newly created object. Note that the system considers any agent holding an owner capability as a legitimate "owner" of the object referenced by that capability (i.e., there may be more than one owner).

An owner can register less powerful capabilities for an object. There are five different rights capabilities may grant over an object: read (R), write (W), execute (X), destroy (D), and *protection domain extension* (PDX), which is explained in Sect. 2.4. Each valid capability grants the holder one or more of these rights to an object.[1] A capability granting RWXD rights is, by definition, an owner capability.

Capabilities are user objects and can be stored and passed around freely. They are implemented as *password capabilities* [APW86], protected from forgery by sparsity. Each capability consists essentially of two parts: the base (64-bit) address of the object the capability refers to (represented as the number of the object's first page), and a (64-bit) password. The password is chosen by the owner when the capability is registered. It is normally obtained from a library routine. Presently, we use a DES-based encryption scheme for creating "random" passwords. However, in the future we plan to use a hardware device producing truly random bitstrings [Wal90]. The list of valid capabilities for each object is maintained by the system in a distributed system-wide directory, the *object table* (OT).

As capabilities are user objects, it is not possible to determine the tasks and users who have access to a particular object. It is also impossible to prevent a particular user who has been given a capability for an object, from handing this capability to other users. However,

[1] Note that, as we rely on the hardware to enforce protection, on many architectures we cannot guarantee that a user cannot read an object to which they only hold an X capability.

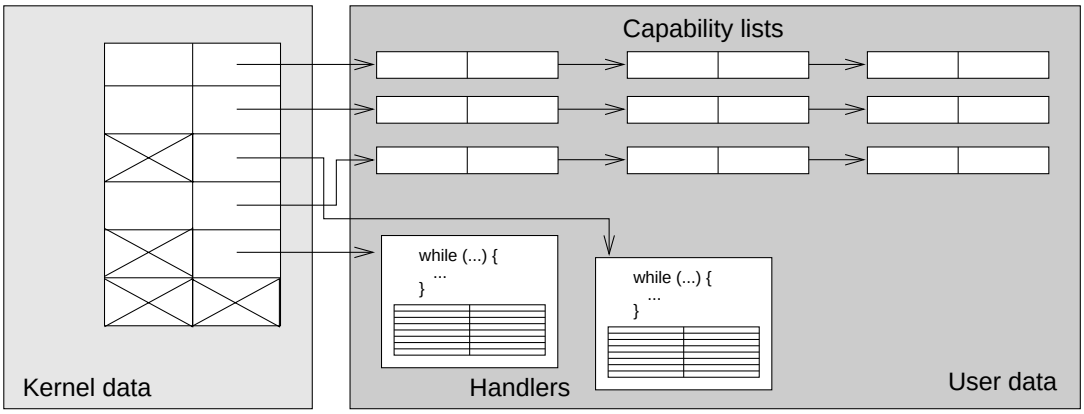

Figure 1: Active protection domain

it is possible to *revoke* a capability completely by de-registering the corresponding password, rendering all copies of the capability useless.

2.2 Objects

An object, once created, persists until explicitly destroyed, and may outlive its creator. To reduce a proliferation of garbage objects, we maintain for each task a *kill list* of all objects created by that task. The object may be removed from the kill list by an explicit system call, allowing it to survive its creator.

The address space released by deleted objects can be reused for new objects. When a new object is allocated in the place of an old one, the use of random passwords ensures (in a statistical sense) that the new object receives different passwords than the old one. Hence dangling pointers and capabilities do not present a security problem.

Address space reuse is important as otherwise even a 64-bit address space could conceivably be exhausted [KC94]. With reuse, address space consumption is essentially limited by the amount of backing store available, which ensures that a 64-bit address space will suffice until it becomes feasible to connect billions of gigabytes of disk to a single system.

2.2.1 Object table

All information about objects, including the set of valid passwords for each object, is recorded in the object table. The kernel (and a few "privileged" tasks) hold capabilities to this table.

To date we have not built a distributed version of Mungi. However, we believe that the following aspects of the design of the OT should allow efficient distribution.

- The OT is based on a B^+-tree, which allows efficient searching for virtual addresses, and can be used to easily partition the virtual address space into separate subtrees which can then be distributed.
- The address space is partitioned, and each node is assigned one or more partitions. Each node can only create objects in its partitions of the address space. This in no way prevents data from migrating to other nodes, but does require that requests to delete an object

are forwarded to its creator node. The creator node plays no special role in any other operations. This strategy ensures that all updates of a particular part of the OT index structure are performed by a single node.

- Some of the object meta-data held in the OT changes infrequently (like the list of passwords). Other meta-data, such as time stamps, do not require strict coherency, and can be updated lazily by an appropriate protocol.

- Descriptors for new objects are entered into the OT lazily. An object is not guaranteed to be in the OT unless it has been made persistent (by performing a system call to remove it from the kill list). This avoids any OT updates for short-lived objects such as most stacks and heaps. The kernel can re-use objects which have never made it into the OT without compromising security, as no other task can access an object which is not in the OT.

2.3 Active protection domains

The main design goal of the Mungi protection system is to be as unintrusive as possible. Applications should normally not have to explicitly deal with capabilities. Consequently, we do not require explicit presentation of capabilities in order to access an object. Instead, the system allows capabilities to be stored in a user-controlled data structure which is searched by the kernel when validating access to an object. This data structure is called a task's *active protection domain* (APD).

The APD, as shown in Fig. 1, consists of a set of *capability lists* (Clists), which are user-level objects conforming to a standard format. The user provides capabilities for these Clists to Mungi, which are then kept in a list in kernel space. In order to support user-defined implementations of Clists for special purposes, the user may also provide addresses of *capability handlers*.

When validating access to an object previously unreferenced by a task, the kernel first finds the object's entry in the OT. The kernel then traverses the APD in search of a capability matching one of the passwords in the entry. When a Clist capability is encountered, the kernel searches the corresponding Clist. If instead it finds a handler address, that handler is upcalled with the object's base address as a parameter and is expected to return a capability for the object (or NULL). If the APD is exhausted without a matching capability having been found, a protection fault is raised.

Mungi provides system calls to allow users to add or remove Clist capabilities or handler pointers from the APD. When a Clist capability is added it is immediately validated before the kernel stores it in the APD. These capabilities are revalidated periodically to detect invalidations by the owners of the Clists (c.f. Sect. 3.2.4). Handler addresses do not need to be validated as the failure of upcalls is not a security issue. This is discussed in more detail in Sect. 3.2.2.

Although the management of Clists is not the kernel's responsibility, we envisage that users will make Clists persistent and group them together to construct a workspace that defines a user's environment. When a user logs on, the APD of their shell will be initialized from their workspace. The majority of tasks they create will inherit this APD and will make few, if any, changes to it. As a result, most applications are unaware of the presence of the capability system.

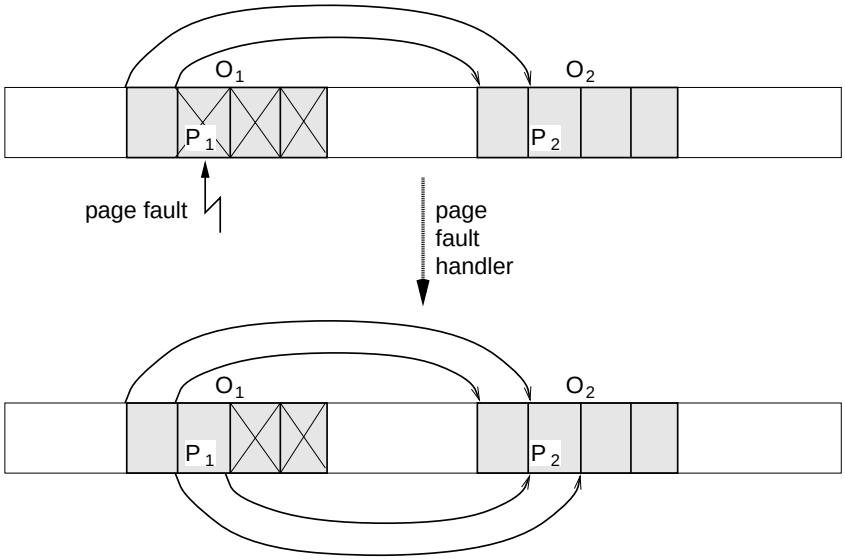

Figure 2: Page fault handling by a user-level pager. Top: A page fault occurs in page P_1 of object O_1. Bottom: The pager has made P_1 resident by mapping it to page P_2 of object O_2 (which, in this case, is handled by the default pager). O_2 is hidden from application code (by keeping its capability secret). Non-resident pages are crossed out.

2.4 Protected procedure calls

In a SASOS, threads normally communicate via shared memory. However, in many cases a more controlled access to data by clients is required—essentially we want a mechanism to support object encapsulation. This can be done if the object in question is not part of any of the potential clients' protection domain, but a mechanism is provided for the clients to invoke methods which operate in a protection domain which includes the object.

One way to achieve this is by having active objects, i.e. objects associated with a server task. Encapsulation can then be achieved by providing a mechanism such as remote procedure call (RPC), which would be used by clients to have the server perform operations on the object. This approach has a number of drawbacks. For example, the server task needs to be running before any client can communicate with it, and its ID must be known to potential clients. This could be achieved by registering the server so that the system will start the server at boot time, and have the server register its ID with some well-known naming service.

A more significant problem is that the client and server often need to share some data, while their protection domains are, in general, disjoint. The client must then explicitly pass capabilities to the server. This conflicts with our goal of providing protection in as transparent a fashion as possible, and incurs the additional expense of validating the capabilities in the server's protection domain. This additional expense could be avoided by allowing by-reference parameters to an RPC call which the kernel would map into the server's view of the address space irrespective of whether or not it holds a valid capability, or by the kernel manufacturing a new capability for the purpose of the RPC. We reject these possibilities since they circumvent the normal protection system, and obscure the protection model. In particular, they reduce the owners' control over their objects, as access could not be reliably revoked.

Instead of encouraging the use of active objects and a client-server model, we are using a protected procedure call mechanism similar to the *profile adoption* mechanism of the IBM System/38 [Ber80]. Our mechanism, called *protection domain extension* (PDX), allows the caller of a PDX procedure to extend its protection domain, for the duration of the call, by the protection domain of the callee [VERH96]. Unlike System/38, our PDX mechanism does not require special hardware.

More specifically, a PDX object's descriptor in the OT contains, for each PDX capability, a list of entry points, and a Clist capability. When a PDX call is executed, the system first verifies that the caller possesses a valid PDX capability and is trying to access an entry point that is valid for that capability. The system then extends the caller's APD by adding the Clist found in the OT, and finally transfers control to the PDX code. When the PDX procedure returns, the PDX Clist (and all cached validation information relating to that Clist) is removed from the caller's APD. Note that for the duration of the PDX call, the calling thread executes in a protection domain different from other threads of the same task; i.e., other threads have no access to the called object (unless they also perform a PDX call to the same object).

Instead of having the PDX procedure execute in a superset of the caller's protection domain, the caller has the option of explicitly supplying an APD when calling the PDX procedure. In this case, the call executes in a protection domain which is the union of the supplied APD with the Clist registered for the PDX object. This gives the caller maximum control over which objects the PDX procedure can access. In particular, an empty APD may be passed to the PDX procedure, which then has no access to any of the caller's data (other than any explicit by-value parameters).

2.5 Virtual memory mapping operations

Whenever an object is allocated, the system uses a default page fault handler to manage paging to a backing store. A user-level page fault handler may be registered for an object. As there is no I/O model in the system, a pager cannot use I/O operations to handle a residency fault. Instead, the pager uses another memory object as its backing store.

To support forwarding of page faults, Mungi provides mapping operations between different regions of virtual memory [ERHL96]. Pages belonging to an object O_1 may be mapped to another object O_2, which causes O_2's pager to be invoked when necessary. Page faults may be forwarded several times until they reach the default pager.

Fig. 2 shows how a page fault is handled by a user-level pager. O_1's pager uses O_2 to provide its backing store. When a page fault occurs for a non-resident page P_1 within O_1, the O_1 pager is invoked. The pager can then map P_1 on a page P_2 from O_2, to provide storage for P_1. If P_2 itself is non-resident, the process will repeat. As soon as P_2 becomes non-resident, the mapping is lost, and P_1 becomes non-resident as a consequence.

Copy-on-write is supported by the default pager (and is really just a special case of a mapping operation). Further uses of these mapping operations are outlined in Sect. 2.5.2.

2.5.1 Implications of aliasing

While copy-on-write introduces aliasing on read-only objects (and is thus harmless [CMS90]), mappings potentially introduce the same aliasing problems as in multi-address-space systems. This seems to defeat some of the advantages of a SASOS. However, as a mapping can vanish at any time (whenever the source page of a mapping becomes non-resident), mapping operations

are only useful for page fault handlers, which effectively prevents (ab)use by normal application code.

No problems due to aliasing exist as long as actual data are always accessed through the same virtual address. This is easily ensured if applications only ever get to see the "top level" object; i.e. the final target virtual pages of a mapping chain, while the source (or intermediate) virtual pages remain private to the page fault handlers. This privacy can be enforced by the pagers if the "backing objects" are kept private to the pagers (i.e. no capabilities are given away). The system discourages any other use of aliasing by not guaranteeing any coherency between aliases.

2.5.2 Controlling I/O

I/O in Mungi is simply implemented by mapping devices into virtual memory, where they can be accessed by suitably privileged tasks (i.e. those holding capabilities to the appropriate addresses).

Mappings can also be used to give appropriate applications control over physical I/O operations. To achieve this, physical memory and disk are mapped into the virtual address space. The application may be given capabilities to portions of the mapped physical memory. As these pages never become non-resident, the application can pin some virtual pages by mapping them to physical memory. A write to disk can be forced by flushing a page.

Similarly, by giving an application a capability to some region mapping part of disk storage, the application can control placement of its data on disk, by mapping its objects to particular pages of the disk. This allows databases, for example, to control their I/O as needed.

3 Implementation of Mungi

Having presented an outline of the Mungi system in the previous section, we now need to show that these abstractions can be build efficiently on a conventional architecture. The details of the implementation are given below, while performance figures are presented in Section 5.

We decided to build Mungi on top of the L4 microkernel [Lie95]. The main reason for this approach was that it would make the Mungi system easier to port between different hardware architectures. We also expected that, by basing our system on a well-designed and optimized microkernel, we would find it easier to produce an implementation which can demonstrate that the SASOS approach can lead to very efficient operating systems.

In spite of using a microkernel we still consider our implementation a "native" one, as we implemented the whole system, including re-writing the microkernel for the MIPS R4600 microprocessor. Furthermore, the microkernel is essentially just an internal interface in our design. It does not provide any functionality which is not required by higher levels of our implementation, so there is no redundancy (which would have existed had we based the design on Mach or a monolithic operating system).

3.1 The microkernel

The main features of L4 which made it suitable for our use are its small size, its very efficient process management and IPC, and the flexible address space model it provides.

While the L4 interface is hardware independent (except for details like the number of registers used for by-value IPC parameters), the actual implementation is not. It is mostly written in assembler, and inherently unportable [Lie95]. Furthermore, there were no 64-bit implementations of L4 available at all. This meant that we had to implement L4 from scratch. In the following, we highlight those features of our L4 implementation that impact on Mungi.

3.1.1 Page tables

The R4600 CPU features a software-loaded TLB tagged with an address space ID (ASID). The TLB contains 48 entries, each mapping two neighboring 4kb virtual pages. We maintain a two-way associative TLB cache for fast TLB miss handling.

On a cache miss, the mapping is obtained from a *guarded page table* (GPT) [Lie93, Lie96b]. The GPT is an efficient data structure well suited for large, sparse address spaces.

The main advantage GPTs have over alternative data structures, such as inverted page tables (IPTs) [CM88, RA85], is that they efficiently support sharing of large areas of the address space. In our implementation we use this for quickly mapping kernel data structures (e.g. a virtual array of thread control blocks) into the client's view of the address space for the duration of a system call. Using clustered page tables [THK95] would have been a possibility. However, we are doubtful as to whether clustered page tables can handle very sparse address spaces, with many single-page objects, as efficiently as GPTs.

Our implementation on the MIPS R4600 CPU takes 1,900 cycles (19 μs) for handling a page fault, i.e. taking the fault and establishing a mapping.

3.1.2 Tasks and threads

A task in L4 is a set of threads sharing an address space. Each task also contains a special thread ("T_0"), which is used for handling exceptions, including IPC events and page faults, on behalf of the task. L4 tasks and threads are very light weight, for example creating a thread takes about 10 μs. Creating a task costs about 75–100 μs (depending on the number of cache misses), while deletion of a task takes about 47 μs.

3.1.3 Inter-process communication

IPC in L4 is designed to be extremely efficient. An IPC call can pass by-value parameters through registers. In addition, it can pass large memory regions by-reference by mapping them into the recipient's address space. As we will show in Sect. 3.2.2, Mungi only uses L4's IPC and address spaces to manage protection domains.

The cost of a null IPC is 91 cycles on the R4600 (compared to the cost for a null system call of 49 cycles).

3.2 The Mungi layer

The L4 microkernel provides a high-performance base on which to build Mungi. Although the L4 interface was not originally envisaged to be used to support a SASOS, its flexibility and simplicity has made it a effective platform for Mungi. The following sections describe the implementation of Mungi.

3.2.1 The Mungi server

The Mungi API is implemented as an L4 user-level server. The main role of the server is to maintain the Mungi attributes of tasks, threads and objects. As well, it is responsible for enforcing the Mungi protection and addressing model.

The server contains a number of threads dedicated to specific events; for example, one of these threads handles Mungi "system calls", which are translated by library stubs into IPC to this thread. Some of these calls, such as Mungi thread operations, which correspond closely to L4 operations, can be forwarded to L4 with minimal overhead.

Mungi uses another one of its threads to act as the default pager for all user tasks. Other threads in the server are used for purposes such as semaphore management and time keeping.

While Mungi makes use of message passing IPC for interaction between these threads, Mungi user threads are not aware of this IPC.

3.2.2 Protection domains

Each Mungi task's protection domain is implemented as a separate L4 task and L4 "address space". The role of these address spaces is to provide separate Mungi protection domains, and their translations from virtual to physical addresses are always consistent with each other to provide the single Mungi address space.

For each protection domain the Mungi server maintains a cache of access validations, consisting of a list of (address range, rights) pairs. This cache is consulted by the Mungi server when handling a page fault. Only on a cache miss will the server perform a full validation, requiring a search for matching capabilities of the OT as well as the APD. Hence, validations normally only need to be performed on the first page fault to a previously unaccessed object.

A new Mungi task can either be explicitly given an APD by its parent or it can inherent its parent's APD. In the latter case, the child will also inherit the parent's access validation cache. Creating a task in this way carries minimal overhead.

Each Mungi task uses the L4 T_0 thread, which is invisible to user code, to handle asynchronous events. For example, L4 translates exceptions into IPCs to T_0 of the appropriate task. T_0 will typically forward the exception to the offending thread.

Upcalls by the Mungi pager thread to a capability handler are implemented as an IPC to T_0 of the faulting task, which executes the handler code. Note that, since the Mungi thread does not execute the handler code itself, the address of the code does not require validation when added to the APD (c.f. Sect. 2.3).

3.2.3 Protected procedure calls

A key concept in Mungi is the use of PDX to provide support for protected procedure calls. PDX is used for device drivers, user-level pagers, and to support object-oriented languages. It is therefore important that PDX be as low cost as possible.

When a thread performs a PDX call, the Mungi server sets up a new L4 task with the extended protection domain. If the PDX call is a proper protection domain extension, i.e. the caller does not provide an explicit APD parameter, the validation cache of the PDX task points to the validation cache of the caller, so the PDX inherits all of the caller's validations.

Once the PDX "task" is set up, the PDX call is translated into an IPC to that task. Exiting the PDX procedure results in a task switch via the Mungi server back to the caller. The PDX

task is then cached by the Mungi server for later calls from the same protection domain. Since the PDX task's validation cache points to the caller's cache, additional validations performed by the caller between PDX calls (or by another thread of the calling task while a thread is executing the PDX) have immediate effect on the PDX as well. This also works for nested PDX calls.

PDX procedures which get passed an empty APD are a special case. The L4 task set up to execute the call can be shared by **all** callers supplying an empty APD, no matter from which protection domain they originate. This means that only one L4 task needs to be cached for PDX procedures which need no access to the caller's data. This class of procedures includes user-level pagers.

Caching also works for PDX procedures which get passed an explicit APD. These start off with an empty validation cache. On a repeated call, a hash of the APD is compared with that of any cached PDX kernel tasks associated with the caller task. If a matching task is found, it is used, otherwise a new task is created.

An alternative to setting up a new L4 task to receive PDX calls would be to actually modify the calling task's page tables in order to extend its protection domain. This modification would need to be reversed on return from from the PDX, which would make PDX calls very expensive. One advantage of our implementation is that repeated calls become very fast as they involve little more than an IPC to the PDX task, a very efficient operation in L4. A further advantage is that other threads in the calling task can continue executing without gaining access to the PDX's hidden data.

PDX procedures may be multi-threaded, with several threads of the same task executing the same PDX object concurrently (possibly using different entry-points). This results in all threads sharing the same extended domain.

3.2.4 Objects

Mungi provides operations for the creation and destruction of objects. L4 itself does not actually provide memory allocation services. Rather, it relies on Mungi to manage the address space, which it does by making use of the L4 mapping operations. Objects are solely an Mungi abstraction, and the Mungi server maintains the free list, disk mappings, validation caches, etc.

Caching of validation data could potentially open a security hole: If an object is deleted, and another object is immediately allocated in its place, validation caching could give the holders of capabilities to the old object access to the new object. We avoid this problem by a combination of two strategies: All entries in the validation cache expire after a time period Δt. As objects are deallocated, their address space is not returned immediately to the free list. Instead the address space is entered into a *stale list*, from where it is moved lazily to the free list, but after a delay of at least Δt. This ensures that no validation data to the old object are still cached. Similarly, the Clist capabilities in the APD are revalidated after at most time Δt.

3.3 Lessons learned

As we had hoped, we found that the SASOS model is indeed easy to implement. The need for large parts of a traditional system has been eliminated, such as the management of file system storage, since this job is done by the swap manager. There is no need to support a separate file abstraction, with its data structures, mappings from file positions to disk storage, etc. Unlike UNIX systems, we do not have to worry about the presence of aliases when shared memory is

used. There is also a potential for simplifications at the hardware level, as virtual caches would not require physical tags.

The addition of virtual memory mapping operations has made it possible to incorporate into the single-address-space model user-level pagers and I/O, and leave, for example, the implementation of stability models to the user level [ERHL96]. This allowed us to build a "pure" SASOS, where virtual memory is the only communication medium between processes.

Since we had to implement the microkernel as well as the higher layers of the system, the question naturally arises whether it was a good idea to base the implementation of Mungi on a microkernel. We believe the answer to that question is a clear "yes", for the following reasons:

- The implementation of Mungi (written almost entirely in C) is easily portable between different hardware architectures (and L4 implementations). As the number of L4 implementations increases, so do the platforms on which Mungi is available.
- The microkernel provided a well-defined interface which allowed us to separate our development efforts. While L4 was being implemented on the R4600 target architecture, development of Mungi proceeded on an L4 implementation on the i486. Once L4 was running on the 64-bit system, the port of Mungi succeeded within around two weeks, in spite of both the microkernel and the Mungi layer being very unstable at the time. With more mature systems, the port would be a matter of days.
- By basing our implementation on a well-designed microkernel, many design decisions for the lowest levels of the implementation had already been made for us.
- We still had the option of modifying the microkernel interface should that have been necessary. However, we found no need to do so.
- As we show in Section 5, layering the system did not result in a performance penalty, as our implementation of Mungi outperforms a commercial operating system.

One of the most encouraging lessons learned is that L4 proved to be a very suitable base for implementing a system quite different from what had originally been envisaged as a typical L4 "client".

To date we have only noticed one drawback of this approach. Programmers who are aware of the fact that Mungi is built on L4 can bypass the Mungi API and call L4 directly. The only problem this is likely to cause is that it prevents confinement, as we cannot control IPC between user tasks. Ideally, all IPC should go through the Mungi server. L4 actually provides appropriate mechanisms to control IPC [Lie92], but at the cost of doubling the number of IPCs required to implement Mungi system calls. Alternatively, it would be possible to modify the L4 IPC code to directly enforce the restrictions required.

4 Related Work

Systems using globally valid names for accessing objects have been, in one form or another, around for a relatively long time. The best known one is probably Multics [DD68], which used a global name space of (segment-name,offset) pairs to identify data. However, individual processes executed in their own private address space. Segments were made accessible to processes by mapping them into the address space, where they could be accessed via a segment number. In general, different processes would map a particular segment to different segment numbers, and

hence virtual addresses, so this approach could not resolve the limitations to sharing imposed by private address spaces. In contrast, a SASOS guarantees that all processes can access a particular memory object via the same virtual address, and hence guarantees the validity of embedded pointers across processes.

Capabilities, as introduced by Dennis and Van Horn [DVH66], provide a true global naming space. Capabilities provide a segmented view of memory similar to that of Multics. Unlike Multics, pure capability systems use capabilities, together with segment offsets, as first class memory addresses. Hence, in such a system all processes agree on the address of a data item, just as in a modern SASOS.

Making capabilities (part of) the lowest level of addressing generally implies building special hardware to interpret the capabilities. Historically, there have been a significant number of systems following this approach: from the earliest commercial capability system, the Plessey 250 (see [Lev84]), via the Cambridge CAP computer [NW77], to the IBM System/38 [HSH81], the Intel iAXP 432 [HLM+82] and the Monads system [RA85,AK85], the last probably being the first ever distributed shared memory system. System/38 and Monads in particular, share much of the philosophy of a SASOS, such as a single-level store, orthogonal persistence, object-based protection, and, in Monads' case, transparent distribution.

In spite of hardware support, many of these systems exhibited poor performance compared to traditional designs. Furthermore, all of these systems suffer from the problem that their dependence on special hardware makes it impossible to take advantage of the latest progress in CPU design. With the rapid appearance of new CPUs, there is a clear disincentive for hardware based solutions.

It is probably fair to say that the IBM System/38 (or AS/400) is the only really successful system of this type, partially a result of the intensive use of microcode as a means to decrease the dependence on specific hardware. However, even that system is not completely hardware independent. At the very least, it requires a tagged memory. It is unclear whether such a system would be viable without the backing of IBM's market share in the traditional commercial computing sector. Furthermore, the AS/400 design does not seem to lend itself very well to distribution.

Hydra [WCC+74] was a software-based capability system supporting a large, flat name space for persistent objects. Hydra can be considered the first microkernel architecture, as it implemented at user level many services which were traditionally part of the kernel. Objects were the basic units of protection and encapsulation. However, the lack of an appropriate hardware base made objects and operations on them too expensive [Lev84]. The Xerox Cedar system [SZBH86] features a single address space to enhance sharing. Protection is not maintained by the operating system, but depends on the use of a type-safe programming language (also called Cedar). Such an approach is obviously unable to support legacy software, and seems to be too restrictive, as it will not work with many of the most popular programming languages. Amoeba [MT86] is a distributed system using sparse capabilities for naming and protecting objects. Capabilities are authenticated by an object's server, which therefore needs to be invoked for every operation on the object.

Grasshopper [RDH+96] is a system specifically designed to support persistence. Its basic storage abstraction is called a *container*, which essentially constitutes an address space. Containers, or parts thereof, can be mapped into other containers. Grasshopper presents a generalized model of address spaces, which can emulate a traditional model, such as UNIX, as well as the SASOS model [LRD95]. However, as the single-address-space view is not enforced

by the system, Grasshopper cannot provide the SASOS guarantee that a specific data item always appears at the same virtual address for the duration of its life time, and thus cannot ensure that data containing embedded pointers can always be shared.

Opal [CLFL94] is a recent SASOS targeted for 64-bit architectures. In Opal, memory segments, threads, protection domains, portals (protected procedure entry points) and resource groups (used for accounting) are all first-class objects, protected by capabilities. In contrast, Mungi only has capabilities for memory objects.

Opal, like Mungi, uses password capabilities, which generally need to be presented explicitly, while Mungi uses implicit presentation. A protected procedure call mechanism is supported which has the caller enter the callee's protection domain. As the two protection domains are, in general, disjoint, capabilities need to be passed explicitly to facilitate sharing. Opal supports two different mechanisms for communications, shared memory and RPC. Mungi, in contrast, provides only shared memory (plus semaphores for synchronization). We believe that this is the most natural and clean approach for communication in a SASOS.

The prototype implementation emulates Opal on top of Mach's UNIX server. This approach naturally has a drastic impact on performance, as discussed in Sect. 5. For this reason, the emulated Opal prototype cannot demonstrate the inherent performance advantages of a SASOS.

Angel [WM96] has very similar goals to Opal and Mungi. Contrary to most SASOS approaches, Angel does not use capability-based protection, nor does it have any explicit protection system built in. Instead, it relies on the ability of an object to be accessed or a service to be named in order to protect it—protection is effectively left in the hands of servers, similarly to Amoeba. Angel, like Opal, provides explicit RPC as part of the model.

While the design is aimed at 64-bit architectures, the prototype was implemented on i486 hardware. It therefore is not faced, and does not address, issues resulting from a huge, sparsely used address space. The Angel prototype is distributed, using distributed shared memory technology. The designers of Angel have studied fault tolerance issues [Wil93] and have demonstrated that full POSIX support, including the difficult fork operation, is possible in a SASOS [WMSS93]. Angel outperforms FreeBSD in some microbenchmarks.

Nemesis [Ros94] is another recent SASOS. It differs from Opal and Angel in that the address space is not distributed, and persistence is handled at the user level. Objects in Nemesis export multiple interfaces, which are combined with closures to provide compile-time type checking. By contrast, object support in Mungi is seen to be largely a programming language issue, with PDX providing the basic support required.

Hagimont *et al.* [HMRS96] argue that application code should not have to deal explicitly with capabilities. Their Arias SASOS, presently under development, hides capabilities from application code and describes all protection in an extended interface definition language. We believe that our approach of implicit presentation of capabilities achieves the same goals while doing so in a fashion more appropriate to a SASOS.

5 Performance

All the performance data reported in this section were obtained on an 100MHz R4600 based SGI Indy workstation with 64Mb of RAM. The R4600 TLB has 48 entries, each mapping a pair of 4kb virtual pages. There are two 16kb two-way set associative caches (one each for instructions and data) with a 32-byte line size. The cache-miss penalty is rather high on the Indy, 34 cycles (0.34 μs) for the first item in a line.

Comparisons with SGI's operating systems used the identical platform running Irix 6.2. Comparisons with Opal are based on published data [CLFL94]. These timings had been obtained on a DEC 3000/400 AXP (133.3 MHz Alpha CPU). According to the SPEC ratings, this machine should be roughly as fast as our Indy (give or take 10–20 %).

5.1 Microbenchmarks

Here we present timings obtained for basic Mungi system calls. These were obtained for repeated calls (presumably hot caches), although some of the figures varied strongly between calls, obviously resulting from cache conflicts.

The Indy's high cache miss penalty was evident in the fact that some figures showed an extremely strong dependence on the exact location of user code and stacks. A repeated PDX call, for example, requires approximately 950 cycles, or 9.5 μs without cache misses. Actual timings, however, varied between 10 and 16 μs, depending on the location of the user stack.

Where possible, we are comparing our timings with those obtained for comparable operations on Irix, and for those reported for Opal. The data are summarized in Table 1, the following sections explain the figures.

Operation	*Mungi*	*Irix*	*Opal*
Null system call	4.6	24	>88
Cross-domain call	10–16	450	133
Thread create	83/48	N/A	N/A
Thread delete	48	N/A	N/A
Task create	500	5,600	650
Task delete	270	1,550	2,300
Object create	60	N/A	315
Object delete	150	N/A	900
Object access	137	N/A	239?
Page fault/map	25	N/A	N/A

Table 1: Microbenchmark timings (in μs). See text for explanations.

Null system call

The cost of a null system call is 4.6μs in Mungi, 24μs in Irix (getpid). No figures are available for Opal. In spite of requiring two IPC operations plus one L4 system call for obtaining the task ID, the Mungi version of this call is more than five times faster than the corresponding call in the UNIX system.

Tasks, threads and IPC

Creating a new thread in Mungi takes 83 μs, which reduces to 48 μs if an ID can be recycled from a thread which has already terminated. In a context where threads are created and deleted frequently (and where consequently this cost is most important) this should often be the case. Thread deletion is the same cost as thread creation with recycling, i.e. 48 μs. No thread times

are available for Opal, and Irix does not presently have a thread interface significantly more lightweight than fork.

Task creation costs around 500 μs in Mungi (800 μs with cold caches), the corresponding fork/exec in Irix around 5,600 μs. The equivalent in Opal is creation and activation of a protection domain, which takes 650 μs. In practical terms, however, the difference is much larger than is evident from these figures: While the Mungi task normally starts off with a hot validation cache (inherited from the parent), the Opal protection domain, once activated, will have to attach segments in order to perform useful work.

Task deletion in Mungi has so far only been measured with a cold cache, it takes 270 μs. The corresponding Irix operation takes 1.55 ms, while Opal requires 2.3 ms.

The cross-domain call mechanism in Mungi is PDX, which costs between 10 and 16 μs. The equivalent operation in other systems is an RPC, which costs around 450 μs in Irix, and 133 μs in Opal.

System	*lookup*	*traversal*		*insert*	*total*
		forward	*reverse*		
Irix 32-bit	7.44	4.77	5.13	4.76	22.10
Irix 64-bit	7.78	6.47	7.49	6.23	27.97
Mungi 64-bit	7.95	6.60	7.71	5.31	27.57

Table 2: OO1 benchmark times (in ms) for the single process version.

System	*lookup*	*traversal*		*insert*	*total*
		forward	*reverse*		
Irix 32-bit/message passing	450.2	5.6	6.8	191.5	654.2
Irix 32-bit/shared memory	479.4	5.8	7.3	211.9	704.4
Mungi 64-bit/PDX	22.6	6.7	7.9	11.1	48.3

Table 3: OO1 benchmark times (in ms) for the multiple process version.

Objects

Object creation (which, by itself, does not allocate any backing store) costs 60 μs in Mungi. Less than one microsecond of that is for the OT update (on a 4-level B^+-tree, which is sufficient to hold at least 32 million object descriptors [GBY90]). Segment creation in Opal using a recycled inode costs 315 μs.

Object deletion in Mungi takes 150 μs, compared to 900 μs in Opal. Neither operation can easily be compared to Irix, which does not seem to support a memory file system.

Opal uses explicit attach and detach operations on segments. An attach followed by a detach takes 478 μs "best case". We assume that the cost of an attach is half this time (which is most likely erring in Opal's favor). Mungi does not feature explicit attach/detach system calls. Objects are made available to a task by inserting their capability into a Clist (an infrequent user-level operation). The operation equivalent to an attach is touching an object for the first time. The handling of such a page fault, which includes looking up its entry in the object table,

looking up a valid capability in the APD, validating the capability, updating the validation cache, and mapping the page to a memory frame costs 137 μs. Mapping a further page of a previously validated object takes only 25 μs.

5.2 OO1

As an approximation to a "real-life" application we implemented the *object operations* ("OO1") benchmark [CS92]. OO1 simulates typical operations in a simple object-oriented database system. Client code invokes a database system to perform *lookup*, *traverse* and *insert* operations on a database.

We have only implemented a subset of the OO1 benchmark, as we were only interested in comparing our use of PDX and the single address space with more traditional approaches. Given the simplifications we have made, it is important not to compare the numbers presented below with data published elsewhere. The results are only meaningful for comparing Mungi with a system running the same code (under comparable conditions). More details on the simplifications we have made to OO1 can be found in Appendix A.

Table 2 shows the results of running single-process versions of the OO1 code, i.e. the database exists in the client's address space and is invoked by normal procedure calls. All runs were repeated 20 times and the averages are reported in the table. The data showed standard deviations of 1–7 % in the Irix case and 0.3–4.0 % for Mungi.

It can be seen that for 64-bit code the performance of both systems is very similar. This is to be expected, as identical code was executed, with no system calls between timer calls. Differences can only occur due to code being allocated at different addresses, which could lead to different patterns of cache misses.

It is evident from Table 2 that 32-bit code executes significantly faster than 64-bit code on the chosen hardware; the difference is about 25 %. This must be kept in mind when looking at the multi-process results. Irix 6.2 does not support 64-bit execution on our platform. We managed to get the single-process version of the code running in 64-bit mode under Irix, but the IPC versions of all Irix code had to be run in 32-bit mode. Hence the Mungi results below include a 64-bit penalty of around 25 % relative to Irix.

We ran the OO1 benchmark (with minimal modifications necessary to enable efficient execution) using different protection domains for database and client. In the Irix version, we used two different implementations of client-database communication: the UNIX System-V message passing interface and the SGI-specific and highly tuned shared memory interface (with semaphores for synchronization). The Mungi version used PDX.

Table 3 shows the results of the performance measurements of the IPC version of OO1. The somewhat inferior performance of the Mungi code on the traversal benchmarks (where only one communication with the database takes place) is easily explained with the penalty from running 64-bit code. In the other cases, lookup (1000 RPCs) and insert (400 RPCs) Mungi outperforms Irix by almost a factor of 20. Comparing the values from Tables 2 and 3 for 32-bit Irix code, it can be concluded that the cost of an RPC in Irix is around 450μs, while the same comparison for Mungi yields 14μs, which is consistent with the figures given in Sect 5.1.

The observation that Irix shared memory IPC does not perform better than SysV message passing is explained by the fact that the amount of actual data passed is very small (around two dozen bytes), so that the cost is dominated by the system call and context switching overhead.

5.3 Summary

The benchmarks show that Mungi clearly outperforms a commercial UNIX operating system on some of the most important basic operations, as well as on an IPC-intensive benchmark of database operations. This shows that the single-address-space approach is not intrinsically less efficient than traditional operating systems, and has a significant edge for certain classes of applications. The microbenchmarks also clearly outperform Opal's published results. Obviously, Opal's performance was partly a result of the platform chosen for the implementation of the prototype. However, we have clearly demonstrated that the PDX mechanism can be implemented with very high performance, and is an inherit advantage of our model, compared to the approach taken by Opal.

6 Conclusions

Single-address-space operating systems present a greatly simplified programming model to applications. This makes them an attractive alternative to traditional systems, particularly where data sharing across processes is important, such as object-oriented databases and persistent programming systems.

In this paper we have shown that such a SASOS can be efficiently implemented on off-the-shelf hardware. Our Mungi system, based on our own implementation of the L4 microkernel on a MIPS R4600 CPU, shows performance figures which significantly outperform a commercial UNIX system in several benchmarks.

The results not only show that SASOS can be implemented efficiently, but also confirm that a well-designed microkernel provides an excellent base on which to build operating systems without sacrificing performance.

Availability

The source code for Mungi will in the near future be made freely available under the terms of the GNU Public License. Check the Mungi WWW pages.

Acknowledgments

The authors would like to thank Jing Pang for performing the benchmark runs under Irix, Ruth Kurniawati for implementing the index structure of the OT, and Paul Ashton from the University of Canterbury for valuable discussions. Chris Amies, Dave Goodall, Fondy Lam, Lester Gock-Young, and Weibin Yuan, all former students at UNSW, contributed to the the project in its eariler stages. The project was supported by grant no ### under the Australian Research Council's Large Grants scheme.

A OO1 Implementation Details

For our benchmarks we used the "small" database (20,000 parts) defined in [CS92]. The lookup operation consists of searching 1000 random parts in the database; the database server is invoked once for each part. The *insert* operation creates 100 new parts in the database and connects each to 3 random parts. The total number of database server invocations is 400 in this case.

The *forward* and *backward traverse* operations start from a randomly chosen part and and follow all parts connected to it up to a depth of seven. Due to the way the database is defined, the forward lookup finds exactly 3,280 parts, while the number of parts found in the backward traverse depends on the starting point. All timings reported in Tables 2 and 3 for that part of the benchmark are normalized to the average number of parts found. The traverse operations are entirely performed within the database server, which is invoked only once for the whole operation.

The OO1 specification requires the client and database server to execute on separate nodes. However, as we do not yet have networking implemented in our system we ran OO1 on a single node. Furthermore, OO1 specifies that caches are flushed to disk regularly. As we are (not yet) interested in I/O performance, but wanted to measure the performance of basic system calls, as experienced by user code, we ignored that specification and instead ran everything in memory.

While running the benchmark on a single node, we nevertheless ran the client and server codes in separate protection domains (except for the "single process" results given in Table 2). In Mungi, this means that the client code invokes the database via PDX calls. In Irix this means that client and server are running as separate tasks communicating via IPC.

The benchmark specifies that, during processing, the database invokes a procedure to return data to the client or obtain further inputs. For simplicity, we did not use a cross-domain call for this, but executed the user-procedure in the addressing context of the database.

In order to ensure a fair comparison we used our own random number generator in the benchmark, hence the actual operations performed are exactly the same across systems. We also had the benchmark do its own memory management to avoid unnecessary interference from allocation strategies. All results are based on hot caches.

Our comparison is actually biased in favor of the UNIX version, as we are using virtual memory addresses as object identifiers. A real database in a traditional system such as UNIX could only do this in combination with pointer swizzling or an indirection via an object table, both of which incur additional overhead. This overhead is ignored in our benchmarks. In a SASOS the chosen implementation strategy is possible without overhead and is the natural way to proceed.

References

[AK85] D. A. Abramson and J. L. Keedy. Implementing a large virtual memory in a distributed computing system. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, volume 2, pages 515–522, 1985.

[APW86] M. Anderson, R. Pose, and C. S. Wallace. A password-capability system. *The Computer Journal*, 29:1–8, 1986.

[Ber80] V. Berstis. Security and protection in the IBM System/38. In *Proceedings of the 7th Symposium on Computer Architecture*, pages 245–250. ACM/IEEE, May 1980.

[CLFL94] 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.

[CLLB92] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1992.

[CM88] A. Chang and M. F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6:28–50, 1988.

[CMS90] C. Chao, M. Mackey, and B. Sears. Mach on a virtually addressed cache architecture. In *USENIX Mach Workshop*, pages 31–51, 1990.

[CS92] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17:1–31, 1992.

[DD68] R. Daley and J. Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, May 1968.

[DVH66] J. Dennis and E. Van Horn. Programming semantics for multiprogrammed computers. *Communications of the ACM*, 9:143–55, 1966.

[ERHL96] K. Elphinstone, S. Russell, G. Heiser, and J. Liedtke. Supporting persistent object systems in a single address space. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, pages 111–119, Cape May, NJ, USA, May 1996. Morgan Kaufmann.

[GBY90] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 2nd edition, 1990.

[HLM+82] P. M. Hansen, M. A. Linton, R. N. Mayo, M. Murphy, and D. A. Patterson. A performance evaluation of the Intel iAPX 432. *Computer Architecture News*, 10(4):17–26, June 1982.

[HMRS96] D. Hagimont, J. Mossière, X. Rousset de Pina, and F. Saunier. Hidden software capabilities. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 282–289, Hong Kong, May 1996. IEEE.

[HSH81] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Symposium on Computer Architecture*, pages 341–348. ACM/IEEE, May 1981.

[KC94] D. Kotz and P. Crow. The expected lifetime of "single-address-space" operating systems. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 161–70, Santa Clara, CA, USA, 1994. ACM.

[Lev84] H. M. Levy. *Capability-Based Computer Systems.* Digital Press, 1984.

[Lie92] J. Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, 1992. Springer Verlag.

[Lie93] J. Liedtke. A basis for huge fine-grained address spaces and user level mapping. In *Proceedings of the 7th European Conference on Object Oriented Programming (ECOOP) Workshop on Granularity of Objects in Distributed Systems (GODS'93)*, Kaiserslautern, Germany, July 1993.

[Lie95] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[Lie96a] J. Liedtke. *L4 Reference Manual*. GMD/IBM, September 1996. Available from URL http://www.inf.tu-dresden.de/ mh1/l3/.

[Lie96b] J. Liedtke. *On the Realization Of Huge Sparsely-Occupied and Fine-Grained Address Spaces*. Oldenbourg, Munich, Germany, 1996.

[LRD95] A. Lindström, J. Rosenberg, and A. Dearle. The grand unified theory of address spaces. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 66–71, Orcas Island, WA, USA, May 1995. IEEE.

[MT86] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29:289–99, 1986.

[NW77] R. Needham and R. Walker. The Cambridge CAP computer and its protection system. In *Proceedings of the 6th ACM Symposium on OS Principles*, pages 1–10. ACM, November 1977.

[RA85] J. Rosenberg and D. Abramson. MONADS-PC—a capability-based workstation to support software engineering. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, volume 1, pages 222–31. IEEE, 1985.

[RDH+96] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris. Operating system support for persistent and recoverable computations. *Communications of the ACM*, 39(9):62–69, September 1996.

[Ros94] T. Roscoe. Linkage in the Nemesis single address space operating system. *Operating Systems Review*, 28(4):48–55, 1994.

[RSE+92] 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.

[SZBH86] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8:419–490, 1986.

[THK95] M. Talluri, M. D. Hill, and Y. A. Khalid. A new page table for 64-bit address spaces. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 184–200, Copper Mountain Resort, Co, USA, December 1995. ACM.

[VERH96] J. Vochteloo, K. Elphinstone, S. Russell, and G. Heiser. Protection domain extensions in Mungi. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, pages 161–165, Seattle, WA, USA, October 1996. IEEE.

[Wal90] C. S. Wallace. Physically random generator. *Computer Systems Science & Engineering*, 5:82–88, 1990.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17:337–345, 1974.

[Wil93] T. Wilkinson. *Implementing Fault Tolerance in a 64-Bit Distributed Operating System.* PhD thesis, Systems Architecture Research Centre, City University, London, UK, July 1993.

[WM96] T. Wilkinson and K. Murray. Evaluation of a distributed single address space operating system. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 494–501, Hong Kong, May 1996. IEEE.

[WMSS93] T. Wilkinson, K. Murray, A. Saulsbury, and T. Stiemerling. Compiling for a 64-bit single address space architecture. Technical report TCU/SARC/1993/1, Systems Architecture Research Centre, City University, London, UK, March 1993.

[WSO+92] T. Wilkinson, T. Stiemerling, P. E. Osmon, A. Saulsbury, and P. Kelly. Angel: A proposed multiprocessor operating system kernel. In *European Workshop on Parallel Computing*, pages 316–319, Barcelona, Spain, 1992.