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
Phone: +61-2-9385-5156
Fax: +61-2-9385-5995
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@us.ibm.com

Appeared in
Software: Practice & Experience, 18(9), 25 July 1998

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 boundaries which make data sharing between processes difficult and expensive in traditional operating systems. They offer the potential of significant performance advantages for applications where sharing is important, such as object-oriented databases or persistent programming systems.

We have built the Mungi system to demonstrate that a SASOS can offer these performance advantages without resorting to special hardware. Mungi is a very "pure" SASOS, featuring an unintrusive protection model based on sparse capabilities, a fast protected procedure call mechanism, and uses shared memory as the exclusive inter-process communication mechanism, as well as for I/O. The simplicity of our model makes it easy to implement it efficiently on conventional architectures.

Our implementation of Mungi for the MIPS R4600 64-bit microprocessor is presented, which is based on our port of the L4 microkernel. Mungi is shown to outperform, in some instances by more than an order of magnitude, two UNIX operating systems, Irix and Linux, in several important operations, such as task creation and inter-process communications, and on the OO1 object-oriented database benchmark. As well, we describe how our approach to key issues in SASOS design provides better performance than other systems, such as Opal. Our experience shows that the SASOS concept is viable, and that a well-designed microkernel is an excellent base on which to build high-performance operating systems.

Keywords: persistent systems, distributed shared memory, capabilities, object invocation, performance

Motivation

For many years, the number of bits available for addressing memory constituted one of the most serious restrictions imposed on programmers. While the introduction of virtual memory made it easier for programmers to deal with the limited amount of physical memory available, even virtual address spaces were too small to address all of the data needed by a program.

As a result, current operating systems provide a large number of different address spaces. In a time-shared system, each process has its own address space containing the memory objects on which the program can operate directly. However, a large amount of the data which programs need to use is outside this address space. This especially includes all *persistent* data; i.e., data whose lifetime is independent of any particular process, and which is generally kept in *files*.

Each file is an address space of its own. A data item within a file is addressed by its position relative to the beginning of the file. However, such an address is different from a normal memory address as it cannot be used by an instruction to access the data. Hence, the system has at least two different addressing mechanisms: Data in virtual memory can be named and accessed simply by issuing its virtual memory address, while data in persistent memory is identified by a file name and an offset, and complex operations are required to make the data accessible.

These non-uniform access mechanisms also significantly complicate the long-term storage and sharing of some types of data. Imagine a program which constructs a dynamic data structure, such as a binary tree. The data structure is composed of a large number of memory objects which have been dynamically allocated, and the different sub-objects are connected with pointers. These pointers are virtual memory addresses whose values are not, in general, under the control of the application program, and depend on memory allocation calls performed previously by the process.

Now suppose the program tries to save the whole data structure in persistent memory so that it can be retrieved later. This creates a serious problem: The pointer values have meaning only within their original address space. When moved into a different address space, they become meaningless bit patterns. If the file is read back by a later execution of even the same program, dynamically allocated memory objects end up at different addresses, and the pointers are invalid. Similar problems can occur if programs attempt to pass data structures via an inter-process communication channel. This inability to share or store pointers, unlike other data types, has been a fundamental limitation of operating systems for many decades.

Two traditional strategies exist for dealing with these problems. One is to convert pointers into position independent references for storage or communication, or convert into a format which contains no pointers. This process is called *flattening*, and must generally be done by the programmer. The alternative is to store pointers in a portable form, then translate them automatically when they are used, a process called *pointer swizzling* [1, 2]. Pointer swizzling is only

possible if the system is able to detect all pointers. This imposes significant restrictions on pointer use, which are generally incompatible with languages like C.

Careful use of shared memory also offers a partial solution. Complex data structures can be shared, but only if the shared memory region resides at the same virtual address for all participating processes, both currently *and in the future*. Reaching agreement on the addresses to share is not always possible if more than a few processes are involved. Further difficulties result from the fact that all objects in the shared address range will have the same protection state, and it is difficult or impossible to allow sharing of just parts of the memory.

While these approaches are of some help, the need to move data between multiple address spaces results in programs that are slower, more complex, and less able to cooperate effectively. These problems could be avoided if all data were put into the *same* address space. It has been pointed out long ago [3] that there are significant benefits to be gained by a uniform treatment of all data, no matter how long its lifetime. This is generally referred to as *orthogonal persistence*.

Large address spaces

An address space that is able to accommodate all persistent objects and to allow easy sharing must be large, much larger than the 32-bit addresses available until recently. This was recognised in IBM's System/38 [4] (now AS/400) and Monads [5], which implemented a large shared virtual address space. These systems offered attractive features such as a single-level store, object-based protection, and, in Monads' case, transparent distribution. However, they rely on the use of custom hardware in order to emulate a large address-space on the processor architectures available at the time.

The advent of 64-bit computer architectures, such as the HP-PA, the MIPS R4000, and the DEC Alpha, has now made the single address space approach feasible [6–8]. A 64-bit address space is big enough to allow the unification of all data on all nodes of a distributed system of thousands of machines. In such a single-address-space operating system (SASOS) there is a single, system wide name for each object — its virtual memory address. Sharing in such a flat, single address space is trivial, as knowledge of the address is all that is required for accessing shared data. By its very nature, the system guarantees that all data is sharable, independent on whether or not sharing was intended at the time the data was created.

The large single address space can also incorporate all persistent data. In a SASOS the address space persists throughout the life of the system, and hence objects allocated in the address space persist as well. As a consequence, a SASOS needs no file system, and non-volatile (disk) store is nothing more than backing store for virtual memory paging.

Eliminating the need for a file system does more than just simplify data storage and application programs. Redundant data movements inherent in file system are avoided, as data read from disk is deposited directly where it is to be

accessed by user code. Soltis [9] also points out that the process independence of virtual-to-physical address mappings in a single address space plays a significant role in keeping context switching costs low. It has furthermore been noted [10] that the simplified model significantly reduces the complexity of the operating system, and leads to improved performance (small is beautiful!) In a distributed system, the single address space incorporating all nodes makes process migration easier to implement: Once a process' context is migrated, data accessed by the process will move as needed.

Recent work

In recent years there have been several projects investigating the design and benefits of SASOS. They have in common a 64-bit distributed persistent memory, which is implemented without the use of specialised hardware. The systems differ, however, in interesting ways.

The Angel [6, 10] project was the first of these systems. The designers of Angel have studied fault tolerance issues [11] and have shown that, by making the distributed single address space fault tolerant, this reliability is automatically inherited by other software structures built on top. They have furthermore demonstrated that full POSIX support, including the difficult fork() operation, is possible in a SASOS [12]. Angel has no explicit protection system. 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. This approach is similar to that taken in the Amoeba distributed system [13], where servers use sparse capabilities for naming and protecting objects. While the design is aimed at 64-bit architectures, the Angel prototype was implemented on i486 hardware. It therefore has not considered issues resulting from a huge, sparsely used address space.

Opal [14] in contrast uses password capabilities [15] to name and protect memory segments, threads, protection domains, portals (protected procedure entry points) and resource groups (used for accounting). Access to these objects generally requires that the correct capabilities be presented explicitly. 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, like Angel, supports two different mechanisms for communications, shared memory and remote procedure calls (RPC).

The prototype implements Opal on top of Mach, and uses the Mach UNIX server for support. This approach has a impact on performance, as discussed below. For this reason, the Opal prototype has not fully demonstrated the inherent performance advantages of a SASOS.

Nemesis [16] is another recent SASOS designed for efficient support of multimedia applications. Its 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.

Grasshopper [17] is a related system. 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 generalised model of address spaces, which can emulate a traditional model, such as UNIX, as well as the SASOS model [18]. 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.

The Mungi system

In this paper we present the *Mungi* system. 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 also decided to take the memory-only model as far as possible, and eliminated explicit support for I/O and conventional message-passing inter-process communication (IPC). An important consideration was not to use any special hardware and implement Mungi on off-the-shelf 64-bit workstations. Our measurements show that this approach works: The system can outperform, by a significant margin, traditional systems if applications make full use of the model.

In the following sections we present first an overview of the Mungi system. This is followed by a discussion of our implementation. We then present our performance measurements which compare Mungi to two UNIX systems as well as another SASOS, and demonstrate Mungi's superior performance, often by more than an order of magnitude.

Overview of the Mungi System

The basic abstractions provided by Mungi are: *capability*, *object*, *task*, *thread*, and *protection domain*. There also exists the concept of a *bank account*, which is used to implement limitations on resource use.

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 [19]. This reduces the need for most applications to deal with capabilities and thus makes protection transparent.

Mungi is a pure SASOS in that it provides no inter-process communication facility other than shared memory (plus semaphores for synchronisation). Furthermore, 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.

It may not be obvious why semaphores are needed for synchronisation, as instructions for user-level synchronisation are provided in most modern architectures. The answer is that in a distributed system these instructions are unusable for synchronisation unless strict memory coherence is enforced. Strict coherence has been shown to be expensive, and unnecessary for many distributed applications [20]. In order to maintain sufficient flexibility until we have gathered enough experience with distribution in Mungi, we decided to support synchronisation in the OS. In a later version this can be moved to a library should that turn out to be sufficient.

The remainder of this section describes in more detail the basic Mungi abstractions. A full description of the API is given in [21].[1]

Capabilities

While SASOS make sharing of data easy, this must not happen at the expense of security. In a traditional OS, memory protection is based on the fact that address space boundaries can only be crossed with the cooperation of the OS, and so access to objects external to a process' address space is under full control of the system. As there are no such address space boundaries in a SASOS, this seems, at first glance, to weaken protection.

In fact, memory protection in SASOS is by no means weaker than in traditional systems [14]. As far as protection is concerned, the concept of an address space is replaced by a *protection domain*, which is the set of objects a process is allowed to access. As in every virtual memory system, a process can only access areas of virtual memory for which a mapping to physical memory has been established, and every attempt to access unmapped memory will result in a page fault. When the system handles that fault, it can verify whether the process has permission to access the memory region; i.e., whether it is part of the process' protection domain. If it is not, the system will generate a *protection fault*. In essence, protection in a SASOS is provided not by controlling what is *in* the address space, but by controlling which parts of it can be *accessed*.

In Mungi, protection domains are defined via capabilities, which 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 which capabilities may grant over an object: read (R), write (W), execute (X), destroy (D), and *protection domain extension* (PDX, which will be explained later). Each valid capability grants the holder one or more of these rights to an object.[2] A capability granting RWXD rights is, by definition,

[1] This document can be found under the Mungi WWW pages from URL http://www.cse.unsw.edu.au/~disy/Mungi.html.

[2] 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.

an owner capability.

Capabilities are user objects and can be stored and passed around freely. They are implemented as password capabilities, 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 [22]. 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 in general impossible to prevent a particular user, who has been given a capability for an object, from handing this capability to other users. However, it is possible to *revoke* a capability completely by de-registering the corresponding password, rendering all copies of the capability useless. Furthermore, we provide the possibility to run untrusted code in a confined protection domain which prevents leaking of data; this is explained later.

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, although they have other problems similar to those of dangling symbolic links in UNIX.

Address space reuse is important as otherwise even a 64-bit address space could conceivably be exhausted [23]. 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.

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, so we have not yet constructed a distributed OT. However, we believe that several features of the

design of the OT should allow for efficient distribution. In particular, the OT is easily partitioned and many of the updates can be performed lazily. This is discussed in more detail in the Work in Progress Section below.

Storage management

While the kill list helps to reduce the amount of garbage objects, this is not enough to prevent all secondary storage eventually filling up with unused objects. Automatic garbage collection does *not* provide a solution [24]. As in a traditional file system, persistent objects are normally entered into a directory, which associates human-readable names with 64-bit object addresses. As long as the directory continues to contain a reference to an object, it cannot be automatically removed as garbage. The system, instead, has to rely on users to manage their storage.

In Mungi no directory services are provided by the system itself. To assist users in managing their storage, we instead use a different, and more flexible scheme, derived from the *rent* model used in Monash University's Password Capability System [15] and the *bank accounts* used in Amoeba [13]. Whenever an object is created, a bank account must be supplied, and the bank account reference is recorded in the object's OT entry. A *rent collector* periodically charges the account for the disk storage used by the object. A *paymaster* periodically deposits funds into each account. An empty or overdrawn account cannot be used to create new objects, forcing the user to clean up. The rent collector issues an *account statement* to show users where their money goes.

An advantage of this approach is that the system is freed from the need to keep track of pointers between objects. The main advantage, however, is that object creation/deallocation is not slowed down by accounting. Accounting is done asynchronously, and the rent collector and paymaster can do their job at times of low system usage. Furthermore, the rent can easily be adjusted in response to high demand, by using a per-byte charge which increases monotonically (and super-linearly) with the total amount of storage used in the system. When disk storage becomes tight, this will force users to clean up. Note that in such a situation users who have let their account balance drop low will be the first to run out of funds, while those using storage economically will be less affected. Such a graceful degradation of service cannot be achieved with a quota system. Details can be found in [24].

It should be pointed out that Mungi's support for bank accounts is limited to requiring the presentation of a valid bank account capability at object creation time. The details of the accounting model are implemented at user-level. Hence it is easy to change the accounting policy to something different than the scheme just outlined.

Active protection domains

The main design goal of Mungi's protection system is to be as unintrusive as possible. Applications should normally not have to deal with capabilities ex-

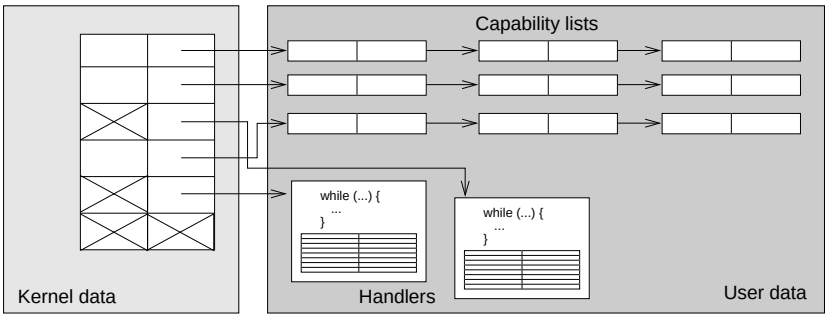

Figure 1: Active protection domain

plicitly. 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 Figure 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, which contains the set of valid capabilities and associated rights for the object. The kernel then traverses the APD in search of the first capability of sufficient strength matching one of the valid passwords. Any Clists in the APD are traversed by the kernel, while capability handlers are upcalled and are expected to return a capability for the object (or NULL). If the APD search fails to provide a matching capability, 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. The kernel can thus trust the Clists to be in the user's protection domain when traversing them. 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 the Implementation Section.

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 initialised from their workspace. The majority of tasks they create will inherit this APD. As a result, most applications are

unaware of the presence of the capability system.

Our use of implicit capability presentation differs significantly from the explicit model used in Opal. To access an object, an Opal task explicitly presents the capability to the kernel, asking that the associated object be attached to the task's protection domain. As well as being less intrusive, implicit presentation as in Mungi provides a significant benefit. There is no requirement that an APD contains capabilities for any of the Clists which define that APD. This provides a mechanism to control propagation of capabilities: A task can be executed in an APD which does not allow access to the capabilities themselves. By providing in addition the possibility to *lock* the APD, preventing further changes, we are able to *confine* an untrusted program [25].

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 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. Tying protection domains to procedures, and performing an automatic change of the protection domain during invocation of such a procedure, is a natural idea in a capability system. It has been used since the 1970s in such systems as the Plessey 250 (see [26]) and the Cambridge CAP computer [27]. Hydra, one of the pioneers of what

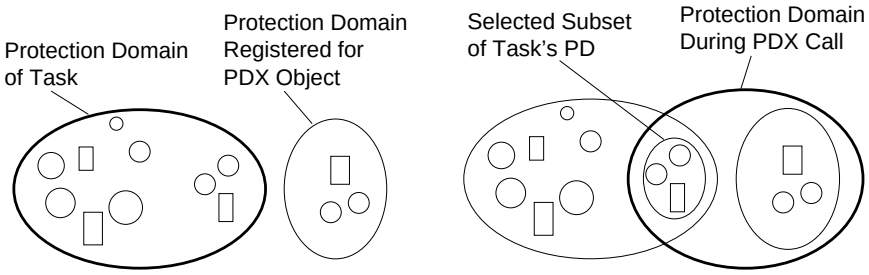

Figure 2: Protection domains before (left) and during (right) a PDX call.

is now called microkernels and the first object-based OS, made protected procedures the basis of secure object invocation. A procedure had an associated capability list which defined the process' protection domain during the execution of that procedure. Sharing of data between the procedure and its caller could be achieved by passing capabilities as parameters, which the procedure could then add to its protection domain. The need to initialise a new protection domain on each procedure call, and to manipulate it again to enable sharing, made protected procedure calls expensive in Hydra.

Our mechanism, called *protection domain extension* (PDX), is similar to the *profile adoption* mechanism of the IBM System/38 [28]. It 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 [29]. Hence, sharing can be achieved without the need to pass capabilities to the procedure via parameters, and changes to the protection domain are minimised. Unlike System/38, our PDX mechanism does not require special hardware, yet allows for efficient implementation (as explained later).

The operation of PDX is illustrated in Figure 2. Prior to a PDX call, the thread is able to access only the objects in its protection domain. During a PDX procedure call, the thread's protection domain will consist of previously inaccessible objects, plus a selected subset (possibly all) of the original domain. After the call, the original domain is restored.

A PDX object's descriptor in the OT contains, for each PDX capability, a list of valid entry points, and a Clist capability. In the simplest case, when executing a PDX call 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 and 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).

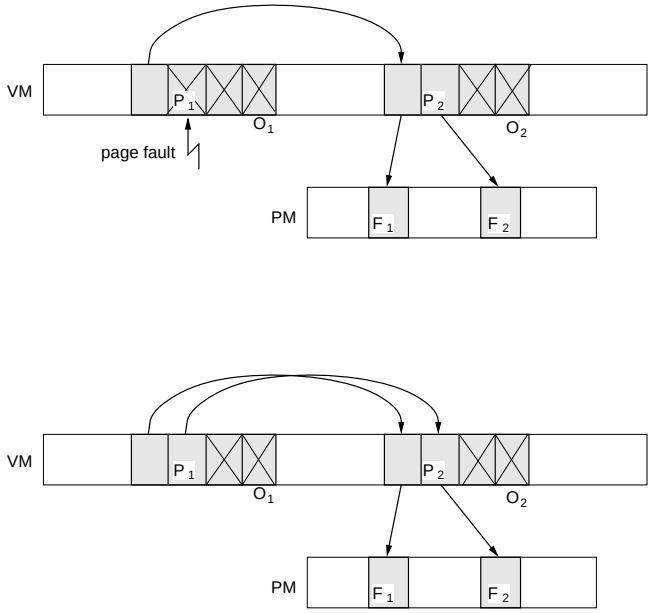

Figure 3: Page fault handling by a user-level pager. Top: A page fault occurs in page P_1 of object O_1. Bottom: The pager has made P_1 resident by mapping it to page P_2 of object O_2 (which, in this case, is handled by the default pager). VM and PM stand for virtual and physical memory, respectively, and non-resident pages are crossed out.

Virtual memory mapping operations

Normally, when an object is allocated, the system uses a default page fault handler to manage mapping of virtual pages to physical frames, and paging them to a backing store. Alternatively, a user-level page fault handler may be registered for the 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 virtual memory object to page to.

To support such forwarding of page faults, Mungi provides mapping operations between different regions of virtual memory [30]. 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 (and thus physical memory).

Figure 3 shows how a page fault is handled by a user-level pager. O_1's pager uses O_2 to provide physical memory for O_1. When a page fault occurs for a non-resident page P_1 within O_1, O_1's pager is invoked. The pager can then map P_1 to a page P_2 of O_2, to provide storage for P_1. If P_2 itself is non-resident, a page fault is triggered on P_2 and the process will repeat until the system default pager is invoked, which maps the faulting page to a physical frame. The mapping of P_1, once established, is lost again as soon as P_2 becomes non-resident.

Controlling I/O

The forwarding of page faults must ultimately result in 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, all physical memory and disks are mapped into the virtual address space. The application may then 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. A similar approach will be used to provide applications with a means to control location of data in a distributed system where this is required for efficiency reasons (normally the single address space hides distribution).

Implications of aliasing

The default pager supports copy-on-write. While this introduces aliasing on read-only objects (and is thus harmless [31]), other mappings potentially introduce the same aliasing problems as in multi-address-space systems. This seems to defeat some of the advantages of a SASOS. Remember, however, that a mapping can vanish at any time, as soon as the source page of a mapping becomes non-resident. This means that mapping operations, while necessary for page fault handlers, are essentially useless for application code.

No aliasing problems 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 if the pagers do not give away capabilities to the "backing objects". The system discourages other uses of aliasing by not guaranteeing any coherency between aliases.

Legacy software and user environments

Unusual or novel computing models are unlikely to find acceptance unless they can support traditional models reasonably efficiently. In particular they must provide user environments similar to established systems.

In the context of a SASOS this means that it must be possible to support applications written for multi-address-space systems like POSIX [32]. Most of the POSIX interface presents no particular problems for emulation under Mungi. A file system, for example, while not necessary in Mungi, can easily be implemented on top of it — it simply provides a file interface to Mungi objects. An open() operation sets up a data structure containing, among others, a current position pointer, which is used by subsequent read(), write() or lseek() operations. No explicit buffering is necessary, so the overhead is somewhat reduced compared to traditional file I/O.

A POSIX open() operation uses the directory system to convert a file's path name into a file location. Similarly the open() emulation in Mungi interfaces to a naming system which converts human readable object names into object addresses. Mungi does not support a system-wide naming facility other than virtual memory addresses; it is left to the user environment to supply a text-based object name space. We presently use an adaptation of the Plan 9 naming system [33], as its concept of user-tailorable naming fits the Mungi model well. However, other views of the object space could be used, including the familiar UNIX naming hierarchy.

One complication arises from to the fact that in the single address space one cannot guarantee that objects can grow to arbitrary sizes — an object can only grow until it hits the next object in the address space. However, the library routines are easily able to hide this. In fact, as long as an object is only accessed via the POSIX interface growth presents no real problem. If an object needs to grow it can be moved to a different location in the address space.[3]

The POSIX protection model can be emulated in Mungi by appropriate organisation of one's Clists [25]. Operations like chmod() are possible in this emulation.

The biggest problem with POSIX compatibility is the fork() system call, which explicitly duplicates the process' address space. This is impossible to do in a SASOS, as there is only one address space. However, POSIX applications which use the fork() operation have a simple view of their address space, which consists of a code and private data and stack segments. There is no need to maintain the integrity of any external references into the data segment. Hence, it is possible on a fork() to create and initialise the child's data segment in a different portion of the address space, provided that the POSIX application is compiled to use suitable indirect addressing [12]. As we shall show later this introduces very little overhead, and this overhead only incurred for applications which actually use fork().

It is worth mentioning that even UNIX applications rarely rely on the full fork() semantics. Usually a fork() system call is immediately followed by

[3] Such a move affects only virtual memory, in reality no data is copied.

an exec() call in the child process. This sequence corresponds well to the TaskCreate() system call in Mungi and can be supported without any problems or overhead.

Other software components, like programming support libraries, tools and window systems, present few problems. Most can be ported to Mungi with little or no effort, but others will require more extensive work. In many cases such ported software will not benefit from any of the inherent advantages of the SASOS model. Nevertheless, they ease migration of users and software.

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 the next section.

We decided to build Mungi on top of the L4 microkernel [34]. The main reason for this approach was that, by basing our system on a well-designed and optimised microkernel, we would find it easier to produce an implementation which can demonstrate that the SASOS approach can lead to very efficient operating systems. We also expected that it would make the Mungi system easier to port between different hardware architectures.

Our choice of the L4 microkernel was also motivated by the close match between our requirements and the L4 model. It provided the basic set of abstractions which we needed to build our system. The implementation of Mungi as a server on top of the L4 microkernel adds a small cost of extra thread switching which might be eliminated in a more integrated design. However, the performance of our prototype shows that the benefit would be small.

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 [35], 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 [34]. Furthermore, there were no 64-bit implementations of L4 available at the time. 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.

Page tables

The R4600 CPU features a software-loaded TLB tagged with an address space ID (ASID). We maintain in software a two-way associative TLB cache for fast handling of hardware TLB misses. On a miss in this software cache, the mapping

is obtained from a *guarded page table* (GPT) [36, 37]. 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) [5, 38], 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 into the client's view of the address space for the duration of a system call. Using clustered page tables [39] would have been an alternative. However, we doubt that clustered page tables can handle very sparse address spaces, with many single-page objects, as efficiently as GPTs. We are still investigating this topic.

Our implementation on a 100MHz MIPS R4600 CPU takes 1,900 cycles (19 μs) for handling a page fault; i.e., taking the fault, scanning the TLB cache, searching the GPT and establishing a mapping.

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.

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 explain below, Mungi uses L4's IPC and address spaces only to manage protection domains. Effectively, Mungi uses L4 IPC to provide parameter passing and context switching between protection domains. The IPC is not used for passing large amounts of data, even though that facility exists. All parameters are transferred in registers.

The cost of a null IPC is 99 cycles on the R4600 (compared to the cost for a null system call of 56 cycles).

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 an effective platform for Mungi. The following sections describe the implementation of the Mungi server.

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 an IPC to this thread. Some of these calls, such as Mungi thread operations, correspond closely to L4 operations, and 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.

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 only 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.

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 upcalls 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 a call 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.

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 first performs a PDX call, the Mungi server creates 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 results in a context switch to that task. Exiting the PDX procedure switches back to the caller's original context. The PDX task is 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) immediately update the PDX as well, eliminating future validation costs. 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 and many device drivers.

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 (as in Hydra). One advantage of our implementation is that repeated calls become very fast as they involve little more than a context switch, an operation which is very efficient operation using L4 IPC. 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.

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: Firstly, all entries in the validation cache expire after a time period Δt. Secondly, 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. As well, Clist

capabilities in the APD are revalidated after at most time Δt. These strategies ensure that no validation data to the old object is still cached when its addresses are reused.

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 and fault tolerance to the user level [30]. 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 provides 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 server being very unstable at the time. With more mature systems, the port would be a matter of days.

- Experience with previous L4 implementations suggested high-performance approaches to issues such as context switching, scheduling, thread creation and destruction etc. This significantly reduced the time spent in developing the lowest software levels.

- As we implemented L4 ourselves we still had the option of modifying the microkernel interface should that have been necessary. The only instance where we found this advantageous is discussed below.

- As we show in the Performance Section, layering the system did not result in a significant performance penalty, as our implementation of Mungi

outperforms UNIX operating systems. This is consistent with the findings of Härtig *et al.* [40], who showed that Linux can be converted to run as a server under L4 at little run-time cost.

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, but this has no effect on the system itself. The only problem that occurs 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 [41], but at the cost of doubling the number of IPCs required to implement Mungi system calls, an overhead which is estimated to be up to ten percent on the faster system calls.

Instead we can support confinement by modifying the L4 kernel such that it disallows IPC between tasks at the same level (i.e., all user tasks under Mungi). There is no run-time cost for this modification, as is shown in the Performance Section.

Performance

In this section we present performance data for Mungi, and contrast them with two UNIX operating systems: Irix 6.2, a commercial UNIX operating system and Linux version 2.1.67[4].

Making direct comparisons between these systems is difficult, as Mungi's superior performance is a result of the use of a fast microkernel as well as the inherent advantages of the SASOS model. However, it has recently been shown [40] that Linux can be run as a server on L4 with essentially unchanged performance. We conclude that whether Linux runs native or as a server on L4 makes little difference in performance.

We also compare Mungi with Opal where possible. This is complicated by the fact that the systems are built on different kernels and different hardware platforms, and by the lack of availability of common benchmarks for comparison. However, we show that the Mungi model provides significantly higher performance for important operations such as cross-domain calls, where Mungi's PDX eliminates the need for capability validation on each call.

All the Mungi, Irix and Linux figures were obtained on an 100MHz R4600 based SGI Indy workstation with 64Mb of RAM. Comparisons with Opal are based on published data [14]. These timings had been obtained on a DEC 3000/400 AXP (133.3 MHz Alpha CPU). According to its SPEC ratings, this machine should be roughly as fast as the Indy (within 10–20 %).[5]

[4] Linux/SGI is available from http://www.linux.sgi.com.

[5] Unfortunately, no exact figure can be given, as we only have SPEC-92 ratings for the Alpha used for Opal, and SPEC-95 ratings for our Indy.

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 20 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 Linux, and for those reported for Opal. The following sections explain the figures, which are summarised in Table 1.

Operation	*Mungi*	*Linux*	*Irix*	*Opal*
Null system call	4.6	6.3	7.7	>88
Cross-domain call	10–20	161	450	133
Thread create	83/48	N/A	N/A	N/A
Thread delete	48	N/A	N/A	N/A
Thread create + delete	131/96	2,450	4,882	N/A
Task create	600	N/A	5,600	650
Task delete	310	N/A	1,550	2,300
Task create + delete	910	34,163	8,150	2,950
Object create	60	N/A	N/A	315
Object delete	150	N/A	N/A	900
Object access	90	45	252	239?
Object create + access + delete	300	447	2470	2,100
Page fault/map	25	N/A	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, its closest approximation in UNIX is the getpid call which costs 6.3μs in Linux and 7.7μs in Irix. A lower limit for the cost in Opal is that of a Mach null-RPC, 88μs. In spite of requiring two IPC operations, the Mungi version of this call is significantly faster than the corresponding call in the other systems.

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 mostly be the case. Thread deletion is the

same cost as thread creation with recycling, i.e., 48 μs. Thread times for Opal were published in [42] for an R3000-based DECstation (create 140μs, delete 230μs). However, as no clock speed or SPEC ratings were quoted for that platform, it is hard to compare these figures. Irix and Linux do not presently have a thread interface significantly more lightweight than fork(), so we used fork()/wait()/exit() as an approximation.

Task creation costs around 600 μs in Mungi (800 μs with cold caches), the corresponding fork()/exec() in Irix around 5,600 μs. In Linux we could not measure task creation separately for lack of a timer of sufficient accuracy. We measured the cost of fork()/exec()/wait()/exit() as the total of task creation and deletion. The surprisingly high task creation cost in Linux (given its generally good performance compared to Irix) might be a result of this version still being under development. In any case, the operations are about an order of magnitude faster in Mungi than in the UNIX systems.

The equivalent to task creation in Opal is creation and activation of a protection domain, which takes 650 μs, or about three times as long as in Mungi.

The cross-domain call mechanism in Mungi is PDX, which costs between 10 and 20 μs. The equivalent operation in other systems is an RPC, which costs around 450 μs in Irix, 160 μs in Linux and 133 μs in Opal.

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 [43]). Segment creation in Opal using a recycled inode costs 315 μs.

Object deletion in Mungi takes 150 μs, compared to 900 μs in Opal. Only the combination of creation, access and deletion could easily be measured in the UNIX systems. The results were about 50 % slower in Linux and eight times slower in Irix.

UNIX systems require files to be opened before first accessing them and closed after the last access. Opal similarly 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 favour). Mungi does not feature explicit attach/detach system calls. Objects are made available to a task by inserting their capability into a user-maintained Clist (an operation that occurs less frequently than subsequent accesses to the objects). The Mungi operation equivalent to an UNIX open() or an Opal attach is touching an object for the first time, which costs 90 μs. This was the only operation we found to be faster in a UNIX system than in Mungi: Opening a file in Linux is twice as fast as validating a first access in Mungi, obviously a result of the simpler UNIX protection model. Irix, however, is much slower.

Mapping a further page of a previously validated object takes only 25 μs in Mungi (no comparable data are available for the other systems).

Summary

These microbenchmarks demonstrate two factors contributing to Mungi's performance: the benefits of an efficient microkernel, and the advantages of the Mungi object and protection model. In particular, the creation and deletion of threads, tasks and objects are significantly faster than in Opal and the UNIX systems. Only on the cost of the first access to an object was Mungi found to be slower than one of the other systems (Linux).

The most impressive result is that for cross-domain procedure calls, i.e., PDX, which is at least an order of magnitude faster than the equivalent operations in the other systems examined. It should be noted that the Opal figures are for a call only, and do not include the validation of the capabilities that are likely to be passed explicitly by the caller. A typical application would therefore see an even greater difference in performance.

OO1

As an approximation to a "real-life" application we implemented the *Object Operations* ("OO1") benchmark [44]. OO1 simulates typical operations in a simple object-oriented database system, which is an example of the class of applications which we believe to benefit most from the SASOS model. Client code invokes a database system to perform *lookup*, *traverse* and *insert* operations on a database. The database needs to encapsulate its data and relies on efficient cross-domain calls for high performance. This benchmark therefore allows us to measure the effectiveness of our PDX mechanism.

We have only implemented a simplified version 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 the Appendix.

System	*lookup*	*traversal*		*insert*	*total*
		forward	*reverse*		
Linux 32-bit	7.99	5.97	N/A	N/A	N/A
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.

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,

0.2–6.0 % for Linux, and 0.3–4.0 % for Mungi. Due to the lack of an accurate timer in the present version of Linux/MIPS, the benchmarks on that platform had to be run with a much increased iteration count. That made the times for the reverse traversal and for the lookup operation impossible to compare with the other results, and they are therefore not given in the table. Also, the Linux benchmarks could only be run in 32-bit mode.

It can be seen that for 64-bit code the performance of Irix and Mungi 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 and TLB 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 and Linux do 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 UNIX IPC versions had to be run in 32-bit mode. Hence the Mungi results below include a 64-bit penalty of around 25 % relative to Irix and Linux.

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 synchronisation). The Linux version used message-passing IPC and the Mungi version used PDX.

System	*lookup*	*traversal*		*insert*	*total*
		forward	*reverse*		
Irix 32-bit/message passing	946	1,445	1445	208	4,047
Irix 32-bit/shared memory	949	1,409	1,411	203	3,972
Linux 32-bit/message passing	344	467	461	842	2,114
Mungi 64-bit/PDX	51	59	61	16	189
Mungi 64-bit/PDX/restricted	50	58	60	16	184

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

Table 3 shows the results of the performance measurements of the IPC version of OO1. Mungi outperforms Irix in average by more then a factor of 20 and Linux by more than a factor of 10. 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, in Linux about 160μs, while the same comparison for Mungi yields 21μs for a PDX call, which is consistent with the figures given in Table 1.

The observation that Irix shared memory IPC does not perform better than System-V 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.

The last line in Table 3 (marked "restricted") was obtained from running Mungi on top of a modified L4 kernel, implementing IPC restrictions. In this version it is impossible to send IPC directly between Mungi user tasks, this version therefore fully supports confinement. As can be seen this is achieved without any run-time penalty, but at the cost of a small modification to the kernel.

Overhead of fork() support

As explained in the Section on Legacy Software, fork() can be implemented by suitable indirect addressing of the data segments, at a cost of reserving one register. To evaluate this cost quantitatively we modified gcc, the GNU C compiler, as done by Wilkinson *et al.* [12]. Table 4 shows the result of benchmark runs which were compiled in three different ways: "normal" compilation (i.e., without support for fork()) using SGI's C compiler, normal compilation using gcc, and compilation using gcc with support for fork(). All versions were compiled for 64-bit with full optimisation. The programs run are the single process version of OO1, and two programs from the SPEC-95 suite. (Note that the times recorded in the table are total execution times, including setup costs and timer overheads, which explains why the OO1 figures are higher than those give in Table 2.)

compilation	*OO1*	*heapsort*	*nsieve*
SGI-cc normal	36.6	39.4	201.8
gcc normal	33.9	37.8	196.8
gcc forkable	34.6	38.0	196.8

Table 4: The cost of supporting fork().

It can be seen that compiling for fork() support results in a run-time penalty of between 0 and 2 %. The biggest penalty was observed in the OO1 benchmark, which is characterised by intensive use of pointers. The penalty for heapsort was 0.5 %, while no difference was observed for nsieve. This is a result of the low pointer usage of these programs, which allows the compiler the optimise away most global pointer uses.

We observe that even in the OO1 case the penalty for forkable code is quite small (and only needs to be borne by programs actually requiring fork()). Note that this penalty is only about one third of what is gained from switching from the SGI C compiler to gcc.

Summary

The benchmarks show that Mungi clearly outperforms UNIX operating systems 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. We even find that the cost of full POSIX semantics, often considered the weak point of the SASOS model, is less than the performance difference resulting from the use of different compilers.

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 inherent advantage of our model, compared to the approach taken by Opal.

The most significant performance advantage of a SASOS, however, will come in areas where the single address space can be used to avoid cross-domain calls or other operating system intervention altogether. This can, naturally, not be demonstrated on small benchmarks, including OO1. We are therefore working on a port of a full-blown object-oriented database system to Mungi, where the potential of taking advantage of the model is particularly great.

Work in Progress

Present work on Mungi is focused mainly on application/user support, and distribution. The former consists of completing the user environment. We are interested in running "real-life" applications which can significantly benefit, in terms of performance, from the SASOS model. As mentioned, we expect object-oriented database systems to belong to this category, and are consequently working on a port of such a system.

Work on distribution centres on providing hooks to applications which require control over data location in a system which hides the network. This is related to the issue of providing control over physical I/O to applications which need it, when the system normally hides I/O. Similarly, the model must allow the specification of coherency properties, or, even better, allow coherency to be implemented efficiently at user level. As Mungi will require different levels of coherency for its own data structure, we are working on schemes which are useful for the OS as well as its clients.

One such data structure is the OT, which we have designed to work efficiently in a distributed environment. In particular,

- the OT is based on a B^+-tree, which allows efficient searching for virtual addresses, and can be used to 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, and
- descriptors for new objects are entered into the OT lazily. An object is not guaranteed to be in the OT unless it has been marked as "sharable" (at creation time or later by performing a system call). 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.

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, as in 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 (Irix 6.2) as well as the free Linux system in several benchmarks.

We have also, as far as possible, compared our results to the Opal SASOS and shown that Mungi offers generally superior performance.

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

Documentation and source code for Mungi and L4/MIPS is freely available under the terms of the GNU Public License, see URL http://www.cse.unsw.edu.au/~disy/Mungi.html for details.

Acknowledgements

The authors would like to thank UNSW students Jing Pang for performing the benchmark runs under Irix, Andrew O'Brien and Conrad Parker for the Linux benchmarks, Luke Deller for his work on the 64-bit compiler for Mungi, and Ruth Kurniawati for implementing the index structure of the OT. We are indebted to Tim Wilkinson and Kevin Murray from City University, UK, and

Paul Ashton from the University of Canterbury, New Zealand, for valuable discussions, and especially to Tim for providing his gcc patches. Chris Amies, Dave Goodall, Fondy Lam, Lester Gock-Young, and Weibin Yuan, all former students at UNSW, contributed to the project in its earlier stages. We would like to thank the anonymous referees, particularly the one who took the time to write several pages of very detailed and constructive comments. The project was supported by grant no A49330285 under the Australian Research Council's Large Grants scheme.

OO1 Implementation Details

For our benchmarks we used the "small" database (20,000 parts) defined in [44]. 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 normalised to the average number of parts found.

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. The specification requires that the database system invokes a user procedure to return data. We implemented this by having the client pass a PDX procedure as a parameter to the PDX procedure used to invoke the database. The database then calls that client-supplied PDX with an empty APD (i.e., a nested PDX call is performed). In Irix and Linux the client and server are running as separate tasks; the client invokes the server via RPC and the server uses another RPC to deliver results back to the client.

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 favour 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

[1] W. Cockshot, M. Atkinson, K. Chisholm, P. Bailey, and R. Morrison. Persistent object management systems. *Software: Practice and Experience*, 14:49–71, 1984.

[2] P. R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *Computer Architecture News*, 19(4):6–13, June 1991.

[3] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26:360–365, 1983.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] F. G. Soltis. *Inside the AS/400.* Duke Press, Loveland, CO, USA, 1996.

[10] 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.

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

[12] 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.

[13] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29:289–299, 1986.

[14] 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.

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

[16] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14:1280–1297, 1996.

[17] 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.

[18] 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.

[19] 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.

[20] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13:205–243, 1995.

[21] G. Heiser, J. Vochteloo, K. Elphinstone, and S. Russell. The Mungi kernel API/Release 1.0. Technical Report UNSW-CSE-TR-9701, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, March 1997. Latest version available from http://www.cse.unsw.edu.au/~disy/.

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

[23] D. Kotz and P. Crow. The expected lifetime of single-address-space operating systems. *Computing Systems*, 9:155–178, 1996.

[24] G. Heiser, F. Lam, and S. Russell. Resource management in the Mungi single-address-space operating system. In *Proceedings of the 21st Australasian Computer Science Conference*, pages 417–428, Perth, Australia, February 1998. Springer-Verlag. Also available as UNSW-CSE-TR-9705 from http://www.cse.unsw.edu.au/school/research/tr.html.

[25] J. Vochteloo. *Design, Implementation and Performance of Protection in Mungi*. Phd thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, July 1998. Submitted.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*, 1990. IEEE Std 1003.1-1990, ISO/IEC 9945-1:1990.

[33] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, a distributed system. In *EurOpen Conference*, pages 43–50, Tromsø, Norway, May 1991.

[34] 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.

[35] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance (still the foundation for efficiency). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Cape Cod, MA, USA, May 1997. IEEE.

[36] 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.

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

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

[39] 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.

[40] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on OS Principles*, pages 66–77, St. Malo, France, October 1997. ACM.

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

[42] M. J. Feeley, J. S. Chase, and E. D. Lazowska. User-level threads and interprocess communication. Technical report 93-02-03, Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195, USA, 1993.

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

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