SCS&E Report 9302
March, 1993

A Distributed Single Address-Space Operating System Supporting Persistence

Gernot Heiser, Kevin Elphinstone, Stephen Russell, Graham R. Hellestrand

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

Persistence has long been difficult to integrate into operating systems. The main problem is that pointers lose their meaning once they are taken out of their address-space. We present a distributed system which has a single address-space encompassing all virtual memory of every node in the system. This design has become possible (and practicable) with the advent of 64-bit microprocessors.

In our system, every pointer retains its meaning independent of its location, even across nodes or on secondary storage. No restrictions are imposed on the use of pointers by application programs. Hence persistence is naturally and elegantly integrated into the system. Further features are uniform addressing and unlimited sharing of data, and memory protection based on password capabilities, making the system easy to use. A reliable paging protocol ensures that the impact of node crashes on other parts of the system is minimised.

1 Introduction

Persistence has long been recognised as an effective means to make applications both simpler and safer. This is due to the persistent system (PS) relieving the application programmer from the burden of having to flatten data structures for permanent storage which, besides the overhead it introduces, eliminates all the protection normally provided by types [1].

The major drawback of persistence is that it is hard to integrate into traditional operating systems (OS). The main reason for this is the fact that persistent objects in a traditional OS tend to change their address during their lifetime. A persistent object is initially created by some process as a memory object within that process' address-space. If that object is stored on secondary memory so that it can later be accessed by another process, it will, in general, be allocated at a different address in the second process' address-space. Therefore the system must be able to find all pointers in persistent objects and translate them appropriately at run-time, or no pointers can be permitted in persistent objects. The latter solution is unacceptable, as it would again force the application to flatten its data structures explicitly before making them persistent, thus depriving the PS of its *raison d'être*. The alternative solution requires imposing restrictions on the structure of (persistent) objects, so that the system is able to find and traverse all inter-object references (as in Napier88 [2]).

While it is possible in such a PS to free the application program from the need to flatten file structures before storing them on disk, the flattening still needs to be done, but now by the (run time) system. Hence the complexity is not removed; it is only pushed from the user into the system domain.

The reason underlying these difficulties is the existence of multiple address-spaces. A persistent object is global in the sense that it can be accessed by many processes. For an individual process to access it, however, the object must be mapped into the process' address-space (and hence in a sense become local). There is only one way to prevent these problems from occurring: to do away with multiple address-spaces, and replace them with *a single, global address-space*.

Such a single address-space OS (SAOS) has been impractical until recently, due to the shortage of address bits. Until about two years ago, standard computer architectures were limited to 32-bit addresses. This was hardly enough for the memory requirements of single processes. To accommodate multiple processes, each process had to be given its own private address-space. The recent advent of 64-bit architectures (MIPS R4000 [3], DEC Alpha [4]) has completely changed the situation. It is now quite feasible to use a single address-space to accommodate all processes in the system; such an address-space is even big enough to incorporate all secondary memory of a computer system, even of a distributed system consisting of hundreds or thousands of workstations. In a SAOS it becomes possible to provide persistence in a clean and elegant way: the requirement of flattening data structures, including the associated overheads in code size and redundant copying, vanishes completely [5].

In this paper we present a *global virtual memory system* (GVMS) featuring a flat, all-encompassing single address-space in which all objects are persistent and potentially visible to all processes. There are no system-enforced limits to sharing, which happens simply by passing around addresses. The system is (almost) purely a software approach, as it will run on standard hardware, possibly augmented by some add-on components to enhance efficiency.

The advantage of combining persistence with a single address-space was recognised a number of years ago by the MONADS group [6, 7]. MONADS features a special architecture designed to support a large address-space. Virtual memory in MONADS is structured to provide hardware support for modularisation and data encapsulation. Sharing is supported by making all modules globally visible, and system protected capabilities are used to provide protection. However, MONADS' dependence on specialised hardware has made it very difficult to let it profit from the rapid advances in computer architecture, as adapting MONADS to state-of-the-art microprocessors is an expensive and time-consuming task.

More recently, Chase et al. [8] have also recognised the advantages of a large, single address-space. Their approach is similar to ours. However, while their publications contain proposals for hardware support for protection, there is little or nothing on such important issues as address-space management, memory coherence, fault tolerance, or even the software model of protection. It is not clear how far these issues have been researched at all.

Another SAOS has been proposed by Carter et al. [9]. That project is also still in a conceptual stage and too little detail has been published to contrast their approach from ours. However, these projects show that the recent progress in computer architecture is leading to a new approach in operating systems design.

The remainder of this paper outlines the basic structure of our system, and highlights the open problems and some of the possible avenues to investigate for their solution. Section 2 contains a general overview. Section 3 describes the proposed memory organisation, while Section 4 discusses the management of the large address space. Section 5 presents the paging protocol proposed to maintain consistency of the address-space in spite of an unreliable communication network. Section 6 discusses management of persistent objects. Finally, Section 7 contains our conclusions.

2 System Overview

The GVMS is a distributed system in which all nodes share the same address-space, so that each byte of data in the system has a unique address. Addresses are (at least) 64 bits wide. No (explicit) system interaction is required to allow two processes to share a data object. If a process wishes to share an object with another process, the former only has to provide the latter with the object's address (and access privileges), after which the second process can access the object in the same fashion as any of its "own" objects.

It is, of course, important to provide protection of objects against access by unauthorised processes. The protection system is based on password capabilities [10], which allows user processes to deal with capabilities as with simple addresses. On the first attempt to use an object, the system validates the process' permission to access the object by comparing the object's password(s) against capabilities possessed by the process. Implicit presentation of capabilities is facilitated by a system-maintained data structure containing capabilities. The capabilities themselves are, however, not system objects and can be passed around freely. Mechanisms for temporarily changing the protection domain (similar to UNIX set-uid) are provided. Details of the protection system are presented in [11].

Distribution is completely transparent to user processes: a process does not know the present location of a memory object, nor does it need to know the location—as soon as an address is referenced, the system will obtain the corresponding page from the network, if it is not already local. The traditional virtual memory model is effectively extended to include the network as well as a local secondary store.

As data can migrate transparently, so can processes. To migrate a process, all that is required is to migrate its process control block (PCB), which includes the register image. On the remote node the registers are reloaded and the process restarted at the appropriate address, after which it will page fault its working set across the network.

Currently we do not plan to support heterogenous processing nodes as we do not consider heterogeneity support as essential to demonstrate the utility of our design. However, we believe that models of heterogeneity developed for related problems [12] can be adapted to work in the GVMS.

Furthermore, our system is not meant to support parallel programming and hence is oriented towards different applications than the *distributed shared memory* projects [13]. We instead aim to support a workstation-like environment of laboratory to building scale (hundreds to thousands of processing nodes). This has implications on the sharing patterns, which are expected to be overwhelmingly read-only or read-mostly, while writable data will tend to be localised on the workstation of their owner.

3 Memory Organisation

3.1 Pages and Objects

Objects are the basic units of memory allocation and protection. The address-space of the GVMS is divided into *pages* of equal size for the purpose of virtual memory paging. Pages are also the basic unit of migration of data around the network. Different pages of a single object may at any given instant reside on different nodes of the distributed system. The latter is hidden from the user as distribution is transparent.

The system's view of an object is a contiguous segment of memory which is page aligned and whose contents have uniform protection state; i.e., any process which is allowed to perform a certain operation on any part of the object has permission to perform the same operation on any other part of the same object. The kernel does not assume or support any structure within objects; higher software layers are however free to impose such a structure.

3.2 Object Table

A single kernel-maintained data structure, the *object table* (OT), contains addresses, sizes, passwords and corresponding access modes of all objects, as well as further data, such as time stamps and accounting information. The OT resides in global virtual memory and is shared by all kernels. The paging protocol (Section 5.1) allows read-only copies of parts of the OT to be cached locally for efficient access.

The OT is structured as a B^+-tree. The interior nodes of the tree form an index, where the keys are the start and end addresses of the objects. The leaves of the tree contain the actual information about each object. Each interior node would contain a few thousand keys, and the height of the tree it therefore expected to be small.

This data structure allows efficient translation of an arbitrary address to its corresponding object (if it exists). Furthermore, the object entries are sorted in address order. This, combined with the partitioning of the address space for allocation purposes, allows efficient distribution of the leave pages of the OT (Section 4).

3.3 Page Table

If an (authorised) process attempts to access a certain memory location, then, as in a traditional virtual memory system, a page fault will be raised if the corresponding page is not in local (physical) memory. In the GVMS, however, there are three different kinds of page faults:

- The page is on local disk. This corresponds to a page fault in a traditional virtual memory system and will lead to the kernel loading the page from disk, after freeing a frame of physical memory.
- The page is not available locally, but resides on another node. This is similar to the basic paging mechanism, except that the page is loaded from a remote node, rather than the local disk. Paging across the network may involve a page ownership transfer as explained in Section 5.1.
- The (virtual) page is not allocated at all, and hence does not reside on any node. The kernel will signal a memory fault to the faulting process.

For recently accessed pages a corresponding entry in the translation lookaside buffer (TLB) will indicate the availability (and location) of the page. For all other cases the kernel maintained page table indicates whether the page is resident, on local disk, on a remote node, or unallocated. A page table for a 64-bit address-space is, of course, much too big to be kept in memory. However, the sparsity of the address space makes it possible to use sparse multi-level page tables, whose pages are themselves being paged to local disk. A subpage of this table, whose entries are all the same (e.g. representing unallocated memory), is left unallocated and the corresponding entry in the higher-level table represents all the entries in the unallocated page.[1]

An alternative approach to page tables could be based on hashing, as in MONADS. We have decided not to use a hardware-based inverted page table to avoid specialised hardware whenever possible. It is still an option, though, to use hashing in conjunction with a software-loaded TLB, as offered by current 64-bit microprocessors.

The page table entries contain, as in a traditional OS, a physical address, dirty, presence, and access mode bits, and a small amount of further data for use by the kernel.[2] The presence bits represent the six cases resident, on-disk (locally), remote, zero-on-use, unallocated,

[1] A consequence of the single address-space is that the address translation information is process-independent and thus does not change on a process switch. This means that there could be a significant gain from moving parts of the page table on-chip, for example the top level table (which might be kept small, say 16–32 entries). This could be a good investment of chip area in future microprocessors.

[2] This includes a type field used to indicate a memory coherency (Section 5.1) or stability model (Section 6.2).

and unknown. In the resident case, the physical address is the frame number of the page in memory. For a page on-disk, the address indicates the disk block. In the case of a remote page, the address field contains a *location hint* identifying the node that is likely to hold the page. In the unknown case, the page is either held by an unknown node or is unallocated.

Zero-on-use pages are useful to support allocation of a large area of memory for growing objects (like stacks or heaps) without wasting a lot of memory for page table pages. A newly created object consists only of such pages. The first time such a virtual page is accessed, a physical page is allocated to it and filled with zeroes, thus becoming a resident page. A zero-on-use page can only exist on the node where the object it belongs to was created.

3.4 Translation vs. Protection

In the GVMS the mapping from virtual to physical addresses differs between computing nodes, but not between processes. Hence, on a process switch, the mapping does not change at all and no translation data need to be invalidated. On the other hand, protection data are process dependent and thus change on a process switch.

If a conventional TLB is used for caching translation **and** protection data, it must be flushed on a process switch, even though all the translation information is still valid. If the TLB entries are tagged with process IDs the invalidation is no longer necessary. However, this implies that for shared pages several TLB entries are required for the same page, one for each of the processes that share the page. This is really a waste of TLB space, as these entries will differ only by a few protection bits.

A possible improvement to this is based on realising that in the GVMS *protection and translation are orthogonal*: translation is associated with pages, while protection is based on objects. If the two concepts can be supported by different hardware mechanisms, significant reductions in process switching costs are conceivable.

Let us assume that, as in conventional systems, a TLB is searched on each address issue for a translation of the virtual to the physical address. In parallel, a *protection lookaside buffer* (PLB) is searched to check the *validity* of the access to the address. The translation information in the TLB no longer needs to be invalidated each time processes are switched. Similarly, the contents of a virtual cache can also be retained. These changes should significantly reduce process switching costs, which are a serious bottleneck in traditional operating systems [14, 15, 16]. Recent work on hardware support for protection in object-oriented systems [17] indicates that a device like the PLB might be possible.

3.5 Page Location

If access to a page of unknown location is attempted, the kernel sends a broadcast message to all nodes on the network. If the page is allocated its owner will reply to that broadcast. For an unallocated page, there is always one node, the one on which the corresponding address-space partition is mounted (see Section 4), which knows that the page is not allocated and will send an appropriate reply to the kernel which tries to locate the page.

In the case of a remote page, where the page table contains a location hint, the kernel attempts to obtain the page by sending a request message to the node indicated by that hint. If the hint is incorrect, the kernel of the latter node will make a broadcast request on behalf of the original requestor, otherwise the page (or a copy) will be sent to the requester. The reliable page transfer protocol is discussed in Section 5.1.

The node pointed to by the location hint may itself possess a (different) location hint for that table. Instead of broadcasting the original request, that kernel could instead forward the request to the node pointed to by its own location hint. This could reduce the number of broadcasts, at the expense of increased latency introduced by the larger number of node-to-node messages exchanged. Further investigations are required to determine if, and how often, a page request should be forwarded.

Locality makes it likely that neighbouring pages are owned by the same node. Hence it seems likely that for a page of unknown status the location hint of the previous page, if available, may lead to the current owner of a page. Again, more research is required to determine the best policy for using location hints of neighbouring pages.

The above scheme ensures that pages which have recently been used (and hence have a valid location hint) can be located quickly. Incorrect location hints will result in unnecessary messages being sent, increasing network traffic and page migration latency. For that reason it is desirable to get rid of old hints. If a page of the page table is brought back into memory, it can be assumed that all the location hints it contains are outdated, and therefore all remote entries in such a page are reverted to unknown. Other timeout schemes for invalidation of old location hints will be investigated in the future.

Note that the page table only contains information about pages which are held locally (either in memory or on disk) and pages which had recently been held locally. The latter set is small due to locality of reference. Hence the amount of information that must be kept in the page table of one node in a distributed system is not much larger than for a stand-alone node with only local (primary and secondary) memory. However, there exists the danger that the page table could become fragmented, so that the *relevant* information, i.e. the entries whose location information is *not* unknown, is sparse within the allocated page table. The extreme case would be that of only one entry per page table page being different from unknown, in which case the page table would by itself fill all local memory. Locality of reference, together with address-space management policies, make this unlikely to happen.

4 Address Space Management

The address-space is statically partitioned; the most significant bits of an address represent its *address-space partition id* (API). The number of partitions is at least as large as the maximum number of nodes the system will support. APIs will be about 10–12 bits long.

Every partition is *mounted* on a single node. Mounting (and dismounting) of partitions is a higher-level management function requiring system administrator intervention.

When a process creates an object, the kernel will allocate memory for that object from one of the partitions mounted on the local node. The pages allocated for that object are *primary* pages of the object and are *owned* by the creator node. Initially, all of the object's pages will be zero-on-use.

Page ownership is a dynamic property; the owner of a page changes as the page migrates around the network. At any instant in time, a page is therefore associated with two (possibly identical) nodes: its *creator* node, which is the node on which the corresponding address space partition is mounted, and an *owner*, which is the node on which the primary page currently resides. The creator is a permanent attribute of a page, while the owner is transient.

For efficiency reasons, nodes other than the owner may hold copies of a page. These copies are *secondary* pages. Secondary pages are always read-only and can only be obtained from the owner of the (primary) page. If secondary pages exist, the primary page is also marked read-only. If an attempt is made by any node to write to the page, all secondary copies are invalidated before the update of the page is allowed to proceed. This write-invalidate protocol is described in Section 5.1.

The creator is the only node which can deallocate the page. Hence, if a process (holding sufficient privilege) wants to destroy an object, the local kernel forwards that request to the object's creator (the node on which the address-space partition containing the object is mounted). The creator obtains ownership of all the object's pages (which implicitly invalidates all secondary pages, cf. Section 5.1), removes the object from the global object table (Section 3.2) and then deallocates the pages.

There are several advantages in partitioning the address space. The major advantage is that OT management is efficiently distributed amongst the nodes. All objects recorded in a leaf page in the OT will belong to the same partition. The node on which this partition is mounted is therefore likely to be the only user of that page of the OT, which reduces contention in accessing the OT. A further advantage of partitioning is that each node can maintain separate free lists for its partitions. This allows efficient creation and destruction of transient objects such as stacks and heaps (Section 6.1), particularly if a separate partition is used for such objects.

No final decision has yet been made on whether memory, once deallocated, can be reused. MONADS never reuses memory, which has the advantage that object addresses are truly unique identifiers for all times. A 64-bit address space is large enough to make this scheme possible. However, our password capability scheme [11] removes the need for strict uniqueness, because new passwords are assigned each time a virtual address is reused. Furthermore, simulations show that reusing memory leads to significantly smaller page tables [18]. These results make reuse of deallocated memory attractive.

The simulations also show that the multi-level page table scheme is feasible. Typically, of the order of 2% of allocated memory is used for page tables, which seems entirely reasonable. The results show that the way address bits are mapped to page table levels is important: From the three schemes in Figure 1, the second results in the smallest page tables while the third one produces the largest.

It must be pointed out, however, that these simulations were based on trace-driven UNIX BSD4.2 file data [19]. Memory usage in a SAOS can be expected to differ somewhat from file usage in a traditional operating system. Hence one should not overestimate the predictive value of these simulations. We do believe, though, that they can be taken as an indication that we are on the right path.

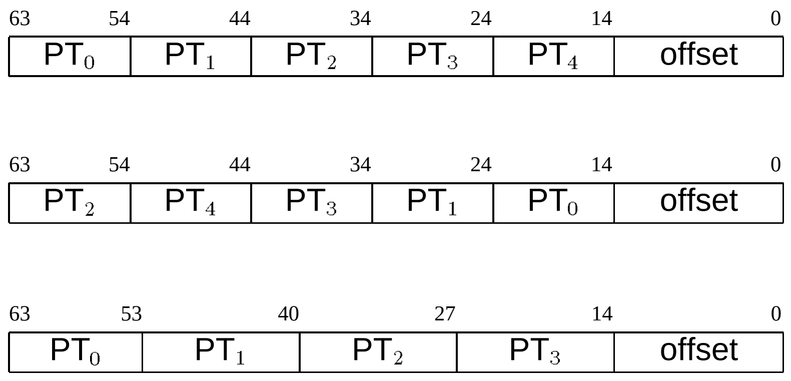
63 54 44 34 24 14 0
PT_0 PT_1 PT_2 PT_3 PT_4 offset
63 54 44 34 24 14 0
PT_2 PT_4 PT_3 PT_1 PT_0 offset
63 53 40 27 14 0
PT_0 PT_1 PT_2 PT_3 offset

Figure 1: Possible allocations of address bits to page table levels. PT_i is the index for the i-th level of the page table

5 Data Migration and Fault Tolerance

5.1 Paging Protocol

As mentioned earlier, every virtual page has a *primary* and possibly several *secondary* physical pages. The node on which the primary page resides is the *owner* of that page. Secondary pages are always read-only, and if they exist, the primary page is (currently) read-only as well.

If a process on node A attempts to read from page P, and P is not resident, A will obtain a read-only copy of P from P's owner B. If the primary page P is writable, B will mark it read-only before sending a copy to A.

If subsequently a process on B attempts to write to P, B will first invalidate all (secondary) copies of P before changing P's status back to writable and allowing the update to go ahead.

If a process on another node, C, attempts to write to the page, C must first obtain ownership of P. To this end, C negotiates the transfer of P's contents **and** ownership. Before transferring ownership, B must again invalidate all copies of P on other nodes (with the exception of C). Note that with this scheme *double faults*, generated by a read-access immediately followed by a write, will introduce extra overhead. This is difficult to avoid with current architectures.

Location of the owner of a page requires a broadcast, unless the page table contains a correct location hint, as explained in Section 3.5. Invalidation of secondary pages may also require a broadcast, with every node required to send an acknowledgement message to the owner. This not only imposes significant load on the network, it also means that invalidation of read-only pages may not be possible if just a single node is down.

Invalidation broadcasts can be avoided if a list of the nodes holding secondary pages is kept by the owner. This is probably impractical, as a large number of objects is likely to be read-shared extensively, e.g. executable code. Assuming that sharing typically happens either between only two participants (e.g. for message buffers), or between a large number (e.g. executable programs), it may be possible to significantly reduce the number of invalidation broadcasts if the holder of a single secondary copy is kept. Such a single node number could be kept in the page table. Further study is required on this issue.

Consistency of the page ownership information is essential for the operation of the system. We therefore use a reliable protocol [20], based on two phase distributed commit [21], for transfer of page ownership. The protocol assures that page ownership cannot be lost or duplicated if messages are lost or duplicated due to an unreliable network or in the case of a node crash. Error detection mechanisms in the network interface are expected to remove corrupted messages.

While the protocol prevents loss or duplication of ownership if a node crashes during the handshake, that exchange will be stalled until both participating nodes, as well as the network connecting them, is operational again. This will in general block at least one process on the receiving node. However, most of the system will remain operational; in particular all processes which only reference local data will be able to proceed unaffected.

To minimise the impact of a single node crash on the remainder of the system, the protocol allows the receiving node to break off the exchange if its original request has not been answered after some timeout period has elapsed. This gives the kernel the chance to signal to a process that some of its memory is temporarily unavailable, and let the process decide whether to retry or to give up. This is expected to be useful particularly for interactive programs.

Implementation of the protocol will require a small amount of stable store on every node. We expect to use a non-volatile RAM (NVRAM) chip for that purpose. The same NVRAM can be used to implement the stable memory needed for critical system data structures (see Section 6.2).

5.2 Memory Coherency

The paging protocol just described automatically implements a strict data coherency model, as it guarantees that (secondary) copies of a page, if they exist, always have exactly the same contents as the (unique) primary page. This could constitute a performance bottleneck if write sharing occurs. However, we do not expect a great amount of write sharing to take place in our system, which is supposed to support a *workstation-like environment*, rather than a *multiprocessor*. In a workstation network most sharing between processes is typically read-only or read-mostly; e.g., via files which are mostly read and occasionally updated.

While our paging protocol guarantees a single writer, it does not provide object-level coherency. Explicit synchronisation must be used to guarantee consistency of objects which are updated by several processes. Assuming that the processor has a test-and-set instruction, we would be able to provide distributed single-bit semaphores for that purpose. However, without further support this might not be very efficient, as testing the semaphore via a test-and-set instruction is really a write attempt which implies acquiring ownership of the appropriate memory page. Some special kernel support for distributed semaphores is therefore likely to be required.

For some applications, our coherency model is unnecessarily strict and introduces significant overhead. We will therefore investigate weaker consistency models based on typed objects [22].

6 Persistent Object Management

6.1 Transient and Persistent Objects

As every memory object created in the system has a globally unique address, which is independent of the process that created the object, every object is automatically persistent. This introduces some management problems for objects which are by their nature not meant to be persistent. For example, a process' stack is an object which has no purpose after the process has exited. The PCB will therefore contain a table of objects that are to be destroyed by the kernel when the process exits. By default a newly created object will be recorded in this table. A separate system call must then be performed to make the object persistent.

The nature of the SAOS will make it easy (and tempting) to create a large number of small objects. As every object must start on a page boundary, this would result in poor memory use. An excess of small objects will also have a negative impact on the performance of the protection system [11]. This will produce a performance penalty for processes using many such objects, which will hopefully discourage their use.

In particular it would be very unwise to create a separate object for each item that would traditionally be `malloc`ed on the process' heap. Instead we will continue to associate a *heap object* with every process, from which normal `malloc` calls will allocate. That heap object will, as the process' stack object, be automatically deallocated on exit. Only objects which are meant to be persistent or which are to be shared with other processes would be allocated separately. Objects which are to be shared but which do not need to survive the process, like message buffers, will be recorded in the PCB so that the kernel will destroy them when the process exits.

Each process is hence associated with (at least) three non-persistent objects: the stack, the heap, and the PCB. The user process has access to the former two, while the PCB is a protected system object. The three objects need to be created each time a new process comes into existence, and must be destroyed once the process exits. Since each object must be recorded in the global object table, this could make processes expensive if the modification of the object table required communication with other nodes. However, the structure of the object table and the partitioning of the address space overcomes this problem (cf. Section 3.2).

While, from the system's point of view, objects have a fixed size which cannot be increased once the object has been created, it is possible to implement growing *user level objects* (like stacks and heaps). To this end a large area of memory is allocated for that object, much more than is likely to be used. This carries no serious performance penalty, as all the virtual pages of that object are initially marked as zero-on-use and are therefore not really allocated. If the object is big, even some of the lower level page table pages will not have to be allocated. Only as the user-level object grows to consume more of the allocated address-space will the virtual pages be allocated in physical memory.

6.2 Stability

Stability of information about objects' existence, i.e. of memory allocation, is essential for the operation of the system. If in the event of a node crash some of this information would be corrupted, different nodes would have differing views of the allocation state of the address-space. Valid object references could cause memory allocation faults, a nasty situation if the reference in question points to important system data. Hence the free list must be stable. User and system objects all live in the same address-space and there is no real distinction between them. Hence the *existence* of all objects must be consistently maintained by the system.

Other data are less critical. Losing some user objects' contents might be painful for the affected users, but the system as a whole would survive such a loss. Hence for a prototype there is no immediate need to provide universal stability.[3] The system can ensure stability of critical data by using an appropriate update protocol.

The main medium used for stabilising objects is, of course, a local disk. To achieve stability, updates on disk must be made atomically. This generally introduces significant overhead, which is not acceptable for all objects, particularly for those which are not meant to be stable in the first place, like local data of a process.[4] There are two possible solutions to this:

Different classes of objects. There are at least two classes of objects: those which are perfectly stable and those who are not. Updates of objects of the former class would immediately be recorded on stable storage. Write access to such objects would be fairly expensive. Objects of the latter class would only be stabilised if the user explicitly asks the system to do so. This gives users control over the effort put into keeping the objects stable, not unlike a traditional file system, where the stable copy of the data may only be guaranteed to be up-to-date after the user performed an explicit close or flush operation on them. As with coherency models (cf. Section 5.1), this object classification is again to be implemented by typing virtual memory.

Caching updates in fast stable store. It has been pointed out in Section 5.1 that some fast stable store, like an NVRAM, is required to record the status of pending inter-node page transfers. This NVRAM can be used more generally as a stable cache for update operations on stable objects. The scheme has the added benefit of improved uniformity: the paging status information is treated just like any other stable object.

Both schemes have their advantages, and the best solution seems to be a combination of the two.

6.3 Garbage Management

Garbage collection in the usual sense is not possible in the SAOS, due to the fact that we do not wish to place any restrictions on the use of pointers by user processes. The system therefore cannot distinguish pointers from other user data, and hence cannot maintain reference counts for objects. As well, garbage collection would likely be impractical owing to the sheer size of the address-space.

Contrary to traditional systems, the need for garbage collection arises not from the necessity to reuse memory (cf. Section 4), but rather from the requirement to reuse the generally much more

[3] Even well-known commercial systems, including UNIX, do not provide stability of user files.

[4] Note, however, that stabilising *all* objects would provide the possibility to save even running processes across system crashes, with the exception of the currently executing one, whose register contents may be lost.

limited backing store for stable objects. Hence it makes sense to clean up backing store rather than virtual address-space. The precise method to use is still subject to further investigation, but will probably be based on leasing [23] or other economic models [24].

6.4 Compilation Issues

The switch to a single address-space will have several effects on the operation of language processors. If it is not possible to generate completely position-independent code, then it will be necessary to assemble the code *in situ*. The assembler allocates an object to contain the binary module, and then all jump instructions are to absolute targets within the object. The linking phase inserts appropriate pointers between binary modules. Execution of the program then occurs in place.

Conventional programming languages can no longer use absolute addressing for static data (such as C's `extern` storage class). This is because in the SAOS absolute addresses are always global and shared by all instances of a program. Private per-process data can therefore only be provided by using relative addressing from a base register. This is not a serious restriction for instruction sets of current architectures, which mostly don't even offer absolute addressing modes of data.

7 Conclusions

We have presented a distributed single address-space operating system which naturally and elegantly integrates persistence. The system features uniform addressing and unlimited sharing in an address-space encompassing all virtual memory of every node in the network. Memory protection is based on password capabilities, which allows user programs to treat all object references like normal memory addresses in a traditional operating system. A reliable paging protocol ensures consistency of the address-space even if the underlying network is unreliable. Proposals for address-space and garbage management and persistent object stability have been outlined.

References

[1] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26:360–5, 1983.

[2] R. Morrison, A. L. Brown, R. Conner, and A. Dearle. Napier88 reference manual. Persistent Programming Research Report PPRR-77-89, Universities of Glasgow and St. Andrews, 1989.

[3] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, 1st edition, 1991.

[4] Digital Equipment Corp., Maynard, MA. *Alpha Architecture Handbook*, 1992.

[5] S. Russell, A. Skea, K. Elphinstone, G. Heiser, K. Burston, I. Gorton, and G. Hellestrand. Distribution + persistence = global virtual memory. In L.-F. Cabrera and E. Jul, editors, *International Workshop on Object Orientation in Operating Systems*, volume 2, pages 96–99, Dourdan, France, 1992. IEEE.

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

[7] J. Rosenberg and J. L. Keedy. Object management and addressing in the MONADS architecture. In *International Workshop on Persistent Object Systems*, volume 2, Appin, Scotland, 1987. IEEE.

[8] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. Opal: A single address space system for 64-bit architectures. In *Workshop on Workstation Operating Systems* [25], pages 80–85.

[9] J. B. Carter, A. L. Cox, D. B. Johnson, and W. Zwaenepoel. Distributed operating systems based on a protected global virtual address space. In *Workshop on Workstation Operating Systems* [25], pages 75–9.

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

[11] J. Vochteloo, S. Russell, and G. Heiser. Capability based protection in a persistent global virtual memory system. School of Computer Science and Engineering Report 9303, University of NSW, Kensington, NSW, Australia, 2033, March 1993.

[12] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3:540–54, 1992.

[13] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[14] J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Conference*, pages 247–56, June 1990.

[15] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, volume 4, pages 75–84, 1991.

[16] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, volume 4, pages 108–21, 1991.

[17] J. Kaiser and K. Czaja. ACOM: An access control monitor providing protection in persistent object-oriented systems. In *International Workshop on Persistent Object Systems*, volume 5, pages 359–73, Pisa, Italy, 1992. Morgan-Kauffman.

[18] E. Kong. Name management and page table organisation for a distributed global virtual memory system. BSc (Hons) thesis, School of Computer Science and Engineering, University of NSW, Kensington, NSW, Australia, 2033, 1992.

[19] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *ACM Symposium on OS Principles*, volume 10, pages 15–24, 1985.

[20] K. Elphinstone. Distributed systems and global naming. BE (Hons) thesis, School of Electrical Engineering and Computer Science, University of NSW, Kensington, NSW, Australia, 2033, 1990.

[21] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[22] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Conference on Principles and Practice of Parallel Programming*, pages 168–176. ACM, 1990.

[23] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on OS Principles*, volume 12, pages 202–10, 1989.

[24] A. S. Tanenbaum and S. Mullender. The design of a capability-based distributed operating system. Technical Report IR-88, Vrije Universiteit, November 1984.

[25] IEEE. *Workshop on Workstation Operating Systems*, volume 3, Key Biscayne, Florida, 1992.