A Password-Capability System

M. ANDERSON,* R. D. POSE AND C. S. WALLACE
Department of Computer Science, Monash University, Victoria, Australia 3168

A tightly coupled multiprocessor is being built in the Computer Science Department of Monash University. This paper discusses several mechanisms that support a uniform virtual memory for the multiprocessor. Some desirable properties of the virtual memory are the result of an unusual capability mechanism and the introduction of a money mechanism. The capability mechanism described also allows a simple, and flexible, solution to the confinement problem.

Received October 1984

1. INTRODUCTION

A tightly coupled multiprocessor is being built in the Computer Science Department of Monash University. The Machine consists of several processors sharing access to a common pool of memory modules. Their common access path is a fast bus of sufficient capacity to support the memory traffic of up to 20 processors of about one Mip.

The machine is not intended for dedicated applications making explicit use of knowledge of the exact configuration of the system. Rather, we conceive the system as being an environment in which many processes may be active and may, if they so desire, communicate and cooperate with each other.

The primary research aim of our project is to ascertain and study the advantages that may accrue from the use of tightly coupled multiprocessors and the impact of difficulties such as congestion arising from competition among the processors for the use of the processor-memory access path.

A generalised virtual memory interface has been devised which defines a uniform virtual memory. All entities in the system communicate through this interface. It uses a somewhat unusual capability-based addressing scheme.

2. THE VIRTUAL MEMORY

The virtual memory and the capability system through which it is used are the bases of our architecture. Although our multiprocessor will be able to accommodate several different processor types, with different instruction sets and different addressing structures, all must operate within the framework of the virtual address space. We have therefore tried to keep the definition of and interface to the virtual memory as flexible and general as possible.

The virtual address space is divided into a number of volumes, each containing a number of objects. Most volumes are associated with single permanent data-storage devices such as fixed or removable disc packs. The association is permanent. A volume cannot be moved from one storage unit to another. Some volumes are associated with a particular multiprocessor system rather than a data storage unit. Again, the association is permanent. Volumes are thus always associated with pieces of hardware. Conceptually, a volume is created when a disc is manufactured or a new multiprocessor is assembled. They are not dynamic constructs. Every volume is identified by a volume identifier, a number assigned to it ideally at the time of manufacture of the device containing the volume and permanently recorded on the device. The volume identifier is unique. Although we do not require that volume identifiers never be re-used, we do require that all extant devices, no matter where located or on what computer they are used, have different identifiers. The management of a numbering scheme meeting this requirement is probably simplest if the identifiers are some sort of manufacturers' serial numbers, in which case identifiers would be historically unique.

Objects are, in general, dynamic constructs. They may be created and destroyed by the actions of processes. When an object is created, it is assigned to some volume and remains in that volume for its life. An object cannot be divided among two or more volumes. A serial number which is different from that of any other object in the same volume is assigned to the object when it is created. It retains this serial number throughout its life. Thus the volume number and serial number together form an object name which uniquely identifies an object among the universe of all extant objects on all hardware devices. The same serial number may of course be assigned to two objects in different volumes. We do not require that serial numbers never be re-used within a volume.

Our design provides for a 32-bit volume identifier and a 32-bit object serial number, which seem adequate for the foreseeable future. An increase in these lengths would have no significant impact on the design apart from a small percentage increase in the sizes of some system tables, and would require no change to the hardware of our present implementation. It would however, be visible at the user level.

With the present design, the volume identifier and object serial number form a 64-bit object name.

3. CAPABILITIES

The virtues of capability-based addressing have been discussed in some detail.[3] Ideally, capabilities can provide security of access with more flexibility than hierarchically nested protection domains, and an unambiguous method of naming objects which does not depend on the context of the processes in which the object is visible.

In any system, capabilities must be protected against accidental or malicious alteration, and must be unforgeable. Two main methods have been used. The first method uses tags on all memory cells and registers to distinguish between capabilities and other information.[4,5,13] The hardware or microcode is designed to restrict the operations which may be performed on cell or register

* To whom correspondence should be addressed.

contents tagged as being capabilities. No unprivileged operation can construct a capability value from other data. This kind of scheme can meet the basic objectives of a capability architecture, but has a number of disadvantages.[1,16] One disadvantage is the space consumed by the tag fields.

The second approach is to segregate capabilities from other data, keeping them in special segments or fenced-off parts of segments.[8,10,18,20] Such segments are often called 'capability lists' or 'C-lists'. Schemes of this kind suffer the difficulty that capabilities cannot be freely treated as items of information to be stored, copied and communicated like other data. In particular, it is difficult to arrange for capabilities to be sent to and accepted from processes outside the computer system, such as processes in a remote machine or human users.

We now describe a capability mechanism that does not require the segregation of capabilities and data and has no tagging.

In our system, a capability is a 128-bit binary value comprising two 64-bit fields. The first is the name of the object. The second is a password permitting certain forms of access to the object. We reserve the term 'capability' for those 128-bit values which do indeed identify an extant object and permit some access to it.

There may exist many different capabilities for the same object. The rights afforded by different capabilities for the same object may be the same or different. Since we define a capability to be a value, rather than a record or other data storage structure, different instances or representations of the same value are different instances, representations or copies of the same capability rather than distinct capabilities. However, different 128-bit values conferring the same rights over the same object are distinct capabilities.

All capabilities for the same object have the same object name but are guaranteed to have different passwords even if they confer identical access rights. Note that the password does not contain a coded representation of the access rights. There is no algorithm for mapping from password values to access rights or vice versa.

We say a capability is created when a value not previously a capability becomes a capability. We say a capability is destroyed when a value which was a capability ceases to be one, i.e. it no longer allows any access to any object. Note that destruction need not involve the alteration of any visible storage cell in virtual memory. No instance of the value need be overwritten or destroyed. All instances of the value simply lose their status and validity as capabilities.

The set of extant capabilities for an object is regarded as part of the object, and resides in the object's volume. If a disc pack containing an object is removed from the multiprocessor and later mounted on the same or a similar machine, all the capabilities for the object remain valid and may be used just as if the object had always resided in that machine.

3.1. Storage and security of capabilities

Because our capabilities are values, they may be stored wherever any data values may be stored, in or out of the computer system. They may be communicated freely between different computer systems or between computers and human users by any means of data communication. They may be encoded into other representations and encrypted.

Our system clearly can given no absolute guarantee of security. Since our capabilities are values, not storage structures, anything capable of generating an arbitrary 128-bit value can in principle get access to any object. We offer only a probable level of security. The probability of guessing a valid capability is very low, even if the name of the object is known. The password field of a capability is 64 bits long. Even if some thousands of valid capabilities exist for an object, the probability of an arbitrary 64-bit value being a valid password for the object is less than 10^{-15}.

The probable security implied by the sparseness of valid password values would be vitiated if it were possible to choose forged values more efficiently than by random selection. That is, we must ensure that knowledge of any number of valid capabilities, together with any other information accessible to a user of the virtual memory, gives no clue as to the values of other passwords. Therefore, whenever a capability is created, our system gives it a password generated by an unpredictable physically random process. The present implementation uses a thermal noise source attached to the system as a shared, high-speed peripheral. A pseudo-random number generator would not suffice, since the output sequences of such generators are inherently of finite algorithmic complexity and hence predictable given enough examples. Note that it is not essential that the sequence of values generated by the random process be free of bias and correlation. Even fairly severe defects of this sort, sufficient to make the generator unsuitable for Monte Carlo and similar calculations, will result only in a reduction of security equivalent to reducing the password size by one or two bits.

The odds of at least 10^{15} against generating a valid password seem more than sufficient to protect against accidental misuse of a capability. Even were users of the virtual memory to make mistaken attempts to use incorrect or out-of-date values as capabilities every microsecond, one would expect to wait about 30 years before any attempt succeeded. It is scarcely conceivable that the innocent mistake rate could be of this order. However, it may be thought that the odds are only marginally secure against a determined attack by a malicious user with knowledge of a valid object name. We have arranged, using the money system described later, that any attempt to use an invalid value as a capability incurs a cost penalty small enough to be tolerable for the occasional innocent mistake, but large enough to make a systematic attack prohibitively expensive in expectation.

A somewhat similar password scheme is used by AMOEBA[15] and, though not in the same context, by Girling[6] to ensure the integrity of objects on a heterogeneous network.

3.2 Master and derived capabilities

Whenever an object is created, a single capability is created for the object and returned to the creator. This capability is called the master capability for the object. Its rights are specified by its creator, and no other right may ever be exercised over the object. Other capabilities may be derived from the master capability or from

previously derived capabilities. To create a derived capability, one must present a valid capability and indicate what subset of its rights are to be attached to the derived capability. All its rights may be passed on to the derivative, but no new one added.

The capabilities for an object have interdependencies which may be modelled by an inverted tree with the master capability at the root, and a derived capability at every other node. The capability at the root of a subtree is an ancestor of all other capabilities in the subtree, and they are its dependants. The immediate ancestor of a derived capability is its parent. The master has no parent or sibling. No capability occurs more than once in the tree. When a new capability is derived from an existing one, the new capability is inserted in the tree as an immediate dependant (child) or the existing one.

The tree structure of capabilities for an object is logically part of the object. It has several effects, the most important being that destruction of a capability implies automatic destruction of its descendants. Destruction of a master capability results in the destruction of all capabilities for the corresponding object, and hence destruction of the object itself, since no further access to it is possible.

The tree structure facilitates unrestricted derivation of capabilities. If unrestricted derivation were permitted without making derivatives dependent on their parents, a kind of rights amplification could occur. For instance, suppose a process X derived a capability A and passed a copy to process Y, which derived from it a capability B with equal rights. In an important sense, B is stronger than A, since it is immune from destruction by process X, which has no knowledge of its existence. The tree, by recording the dependence of B on A, prevents B from becoming stronger than A.

A single-rooted tree ensures that only one capability (the master capability) need be examined to determine the status of an object.

Since it is very easy to share a capability, i.e. the same bit pattern in many locations can represent the same capability, the need to create and distribute additional capabilities for an object is reduced. Thus, the size of the tree of capabilites for an object is expected to be considerably smaller than if it were implemented in a system with either a tagged or segregated capability scheme.

The space used to represent a password capability in a system table is of the order of eight words.

4. OWNERSHIP AND GARBAGE

A problem that faces capability-based architectures is the identification of objects that are no longer needed or are inaccessible. Such objects are defined as garbage.

Most solutions, to identify and remove garbage, use some combination of an ownership scheme, reference counts, and reference tracing.[8, 14, 18]

The basis of an ownership scheme is to mark specific capabilities for every object as 'owner' capabilities. If all owner capabilities for an object are destroyed then the object is defined as garbage. It is interesting to note that ownership schemes were rejected in the HYDRA system design on philosophical grounds.[19]

Schemes based on reference counts involve keeping a count of the number of capabilities that refer to an object. If the count drops to zero then the object is defined as garbage. However, circular references can occur, thus rendering an object inaccessible while still having existing capabilities for it. Any garbage collection mechanism must be able to detect this condition and remove, as garbage, an object that falls into this category.

Our virtual memory involves no concept of ownership. No object is dependent on any other object. Of course an object may contain a capability for another object, and hence effectively point to it, but the capability may be represented in any kind of code and its representation may be unrecognisable as a capability. Hence the behaviour of the virtual memory cannot depend on the existence or structure of any such pointers. Further, records of valid capability values may be held outside the multiprocessor system. It is clearly impossible for the virtual memory to determine if an object has become inaccessible. Hence our system can use neither of the above schemes to identify garbage.

We have addressed this problem in two ways. First, an object is destroyed when no capability value for it is defined, since there is then no way of referring to the object or creating a new capability for it. More interestingly, we have incorporated a charging system into the design of the virtual memory. The charging system was originally introduced as a solution to a resource allocation problem, but was found to be so useful that it has been expanded into a cash economy defined as an integral part of the multiprocessor architecture. Every object is charged a rent depending on its size, and an object is defined as garbage if it cannot pay its rent. Since the charging of rent, and the sources of funds from which rent can be paid, are included in the definition of the virtual memory, the virtual memory can automatically dispose of garbage defined in this way. The money mechanism is further discussed below, but note in this context that it does not introduce a concept of ownership of objects. The rent for an object is claimed from the object itself, not from any owner or 'user code account'.

5. MONEY

Money is a quantity which can be held in an object and transferred from one object to another. It can be viewed as a generalised, transferable right to use any of the services provided by the virtual memory or other system services. Unlike other rights, it is consumed when the service is provided. Money is not an alternative to the rights provided by a capability. Charged services will be provided only when an appropriate capability is provided and money is paid.

Every object can, and must, contain some money. Sums of money are not normal data items and cannot be copied or processed. Money obeys a conservation law: if money is transferred from A to B, the sum at A is decreased by the same amount as the sum at B is increased. The money held in an object is distinct from any data, code, capability or other information stored in the object. Access to the money in an object or to information about the money requires a capability with an appropriate right.

A monetary value is associated with every capability. In the master capability, the value represents the sum of money held by the object. In a derived capability, the value is not a further sum of money, but shows how much

of the object's money can be withdrawn using the derived capability. Withdrawals and deposits using a derived capability are treated as being made via its parent. Thus withdrawal of a sum using a derived capability requires that its money value and those of all of its ancestors including the master at least equal the sum to be withdrawn. A permitted withdrawal or deposit decreases or increases the money values of the capability used, and all of its ancestors. It is not required that the money value of a capability equal the sum of the money values of its children: it may be higher or lower than this sum.

The money mechanism allows our design to avoid some common problems not by solving them but by redefining them. The parallel garbage collector found in other systems[14, 18] has, in this system, been transformed into a rent collector. The rent collector periodically scans volumes to inspect the master capability for each object and deduct rent from its money. Any object whose money is exhausted is regarded as garbage and destroyed. Thus, by defining garbage as bankrupt objects, we make it unnecessary to find all references to an object or to keep reference counts. Similarly, the money mechanism may be used to assist in the management of various resources. For example, consider the case of a process X that possesses a capability for a line printer. By creating and revoking derived capabilities, and receiving payment using the capabilities, *X* can rent out the printer. If rent is not forthcoming then the appropriate capability can be revoked and the next resource request attended to. The objective of *X* is not to ensure that the line printer is used, but to ensure that it is paid for. *X* need not be concerned about processes that become deadlocked using the printer. By transforming *X*'s method of management for the printer, i.e. from ensuring the printer is used, to payment of rent for the printer, responsibility for deadlock avoidance is shifted to the user.

6. PROCESSES

We define a process as a single sequence of instruction executions and virtual-memory interface calls, performed above the virtual-memory interface level. A process has no internal parallelism, and can run on at most one processor at a time. There may be different types of processes running in the multiprocessor, and different types may require to run on different types of processor, but a process may run on any available processor of suitable type, and does not remain tied to any one processor. Sequences of instructions executed to support parts of the virtual memory, e.g. to manage page swapping, physical storage allocation and processor scheduling, are not regarded as processes.

A process is represented in virtual memory by an object which contains, inter alia, the following.

(1) A process type code, determining the type of processor required by the process and the format of the rest of the object.

(2) A processor state, including at least a program counter value comprising a capability for an object containing code and an offset.

(3) A sum of money ('cash') from which payment is made for services used by the process. This sum is additional to the sum associated with the master capability of the object, and is more easily accessible to the processor on which the process is running. The process may transfer money between its cash and the sum associated with its master capability. It has the responsibility of ensuring that it has sufficient cash to pay for the services it uses, including processor time.

(4) A buffer ('mailbox') for the receipt of messages.

(5) The capabilities and view-defining information associated with any window registers used by the process (see section 10).

(6) A 63-bit lockword (see section 7).

All references to a process require use of a capability for the object representing the process. We therefore call this object the process, and may regard a process simply as an object capable of autonomous behaviour. The master capability of a process has rights which allow the object to be viewed as a process.

7. CONFINEMENT

There has been a certain amount of interest concerning the containment of information.[9, 12] Several writers have described mechanisms, some based on capabilities, for allowing a process to call on a software package to perform some service with the assurance that the called package cannot capture or transmit to an unauthorised receiver any information about the data passed to it.[2, 11, 17]

The password capability system permits a flexible implementation of the desired information containment, based on a simple encryption of passwords.

First, we distinguish between capabilities which are strictly read-only and those which permit some kind of alteration of the state of the addressed object (alter capabilities). One bit of the password field of a capability is set to 0 if it is read-only, or 1 if it is an alter capability. Thus, only 63 bits of a password are in fact chosen randomly.

Every process contains a 63-bit 'lockword' which is not readable by the process. Whenever the process creates a capability, either by creating a new object or by derivation from an existing capability, the value returned to the process contains the true password P if the created capability is read-only. If it is an alter capability, the value returned contains an encrypted password Q, being the exclusive OR of the password and the process lockword L. $Q = P \oplus L$. Whenever a process tries to use an alter capability, the virtual memory interface decrypts its password by exclusive OR with the process lockword before checking its validity.

Normally, most processes will have a zero lockword, so the passwords they see are not encrypted. However, if a process P1 with lockword L1 wishes to use an untrusted package, it will create a new process P2 to execute the package. In creating P2, P1 may specify an arbitrary lock value V to be applied to P2. P2 is then created with lockword $L2 = L1 \oplus V$. Before passing any alter capability to P2, P1 encrypts its password by exclusive OR with V. The password known to P1 is already encrypted by L1, so this operation creates a password encrypted by $L1 \oplus V = L2$. Hence P2 may successfully use the passed capability. Further, if P2 creates any new object, the alter capabilities it gets for the object are encrypted with L2. If P2 returns such a capability to P1, P1 may partially decrypt it by exclusive OR with V, giving a password encrypted by L1. P1 may then use the capability to refer to the returned object.

P2 may alter any object it creates, or for which it is given an alter capability by P1. It cannot alter any other object, even if it knows an alter capability, since it has no way of discovering its lockword L2. Note that even if P2 knows both an (L2-encrypted) alter capability and an (unencrypted) read-only capability for the same object, it cannot deduce L2, since the true passwords of these capabilities are not related. P2 may, however, use read-only capabilities built into the code of the package, e.g. to create additional processes, to use other packages, or to access databases.

Although P2 may create objects in which it can place information, it cannot convey their capabilities except by placing them in the objects to which P1 has given it alter access or objects which it has created.

P2 may itself use packages which it does not trust by creating processes under further locks.

Since a process is required explicitly to nominate the lock V to be applied to any process it created, and to encrypt any alter capability it passes to the new process, a process P1 may create two or more processes P2, P3, etc. running under the same lock. Such processes can then freely exchange alter capabilities with each other, but are still contained as a group. More generally, P1 can create processes P2, P3, etc. using different locks V2, V3, etc. If P1 tells P2 the value V2⊕V3, then P2 may encrypt alter capabilities to pass to P3, and decrypt ones created by P3.

There is no file access operation per se, all files being objects in virtual memory paged in as required. Accounting is done by the money system, the movement of money requiring alter capabilities. Interprocess communication via shared memory or via messages also requires alter capabilities. Hence there are few if any covert channels available for violating confinement.

The confinement scheme has been described in the context that the calling process creates a new process to provide the service. It is also possible to use the scheme when the service is called as a procedure rather than as a process. In this case, the process exclusive ORs its own lockword with a nominated V as it enters the procedure. To enable the original lockword to be recovered on exit from the service procedure, the architecture of the process type must provide a stack of lockwords in the process object rather than a single lockword.

8. RIGHTS AS VIEWS

A right conferred by a capability may be regarded as a right to see some specified aspect of an object. That is, it defines a view of the object. The various rights which are recognised in our system may usefully be classified by the kind of view they permit.

8.1 Money rights

These permit an object to be seen as a store of funds. Any object may be so regarded.

(1) Drawing right: the right to draw up to a certain amount of money from the object. It is depleted by use and augmented by deposits.

(2) Balance: the right to enquire what sum can be withdrawn from the object by use of this capability. The result may be less than the drawing right, being the minimum of the drawing rights of this capability and all of its ancestors.

(3) Deposit: the right to deposit money in the object. Deposits require a specific right as they allow a process to convey information and could be used to violate an intended confinement of information.

8.2 Window rights

These allow an object, or part of an object, to be seen as a set of consecutively numbered 32-bit words. The master capability of any object includes a window right for all of the object that can ever be seen in this way, and any window included in any derived capability sees a consecutive subset of the words in the master's window. No capability has more than one window right. The words seen through a window are numbered from zero up, no matter what the number of the first word when seen through the master's window. When a capability is derived from another, the derived window must of course be a subset of the original window, and may start and end anywhere in the original window. Every window right is defined by an offset and size, giving its start and end as seen through the master's window. Specific access rights which may be associated with a window are read, write, and execute.

8.3 Process rights

These provide a view of an object as a process.

(1) Message: the right to send the process a message, and hence to awaken it if it was waiting for a message. A message is a 16-word record of arbitrary content.

(2) Suspend: the right to suspend and resume the process.

(3) Status: the right to find out some details of the internal state of the process.

(4) Condition: the right to initialise a suspended process. This right is exercised in order to perform operations that partially define the state of a process. For example, the setting of the process's program counter.

8.4 Suicide right

The suicide right permits a capability to be used to destroy itself. Consequently, the capability's dependants are destroyed. Suicide by the master capability for an object implies destruction of the object. Suicide right may be given to a derived capability even if the capability from which it is derived does not have this right. This ability may appear to go against the general rule that no capability can have greater rights than the one used to create it. However, a capability's existence depends on the existence of all of its ancestors, whether the capability has suicide right or not. Thus, no amplification of rights takes place.

9. CAPABILITY REGISTERS

This section describes an aspect of our implementation of the architecture. It is included to suggest that a reasonably efficient implementation is possible, but we do not claim this to be the only or best possible approach.

An implementation of our architecture must allow access to the virtual-memory interface. It may also provide constructs which are conceptually above the virtual-memory interface which add to its efficiency or ease of use.

In the normal course of its execution, a process may wish to exercise a window right very frequently, i.e. at memory speeds. It would be intolerable if the process were required on every occasion to present a full word or byte address in the virtual-address space, since a virtual address comprises a 128-bit capability and an offset-within-window which may be as large as 28 bits. A processor could therefore provide capability registers which may be used as addressing bases. A process may load a capability into a capability register, and thereafter refer to words in the window defined by the capability by some sort of logical address which implicitly or explicitly names the capability register and gives an offset relative to the beginning of the window. Similar capability registers are used in other architectures.[1,3]

As will be seen below, our design recognises the existence of capability registers, but their number, organisation and logical addressing may vary from one process type to another.

Strictly speaking, a capability register (or window register as we call it) exists above the virtual-memory interface, and its existence and use is no part of the virtual-memory design. A window register is, in concept, simply a device which a process may use conveniently to construct a full virtual address, which is then presented to the virtual-memory interface for interpretation as a capability-offset pair. However, in our design the production of a virtual address using a window register and the interpretation of that address to yield a hardware access path to the designated word can be collapsed into a single step. The system-wide standard mechanism to support such operations is described in the next section. A particular processor design may elect to use this mechanism or not. For instance, our first model of processor, based on a National 32032 microprocessor, uses the single-step window-register mechanism for addressing words in windows, but not for some operations on processes.

10. VIEWS

Given a capability, it is not a trivial matter to interpret it to obtain access to an object. The capability itself contains little information about the location of the object (only the volume identifier) and no information about the kind of view and access rights it provides. Thus, if only the capability is known, it is necessary to consult system tables to establish the validity of the capability, its right to permit the kind of access requested, and a hardware route supporting the access.

Our implementation has a large table, the Active Objects Table (AOT), which inter alia acts as a main-memory cache for such information. It is addressed by a hash of the object name. None the less, given a window capability, several memory accesses are needed to find all the information required to support a word access. This order of overhead is tolerable for infrequent and complex forms of reference to an object, such as the suspension of a process, but is hopelessly inefficient if incurred on every reference to a word in a window.

The design provides a system-wide interface to a level below the virtual-memory interface. Using this interface, a processor executing a process may present a capability and obtain from the AOT the access rights and location information needed to implement the view defined by the capability. The processor can then perform whatever access to the object was required by the process it is executing. In order that the time-consuming references to the AOT needed to acquire this view need not be repeated for every simple access, we permit the processor to retain the view-defining information in association with a window register. Thus, when a process loads a capability into a window register, the processor executing it obtains the corresponding view using the low-level interface and loads the view information into an extension of the window register. When subsequently the process uses the window register to address virtual memory, the processor can directly check the validity of the required access and generate a hardware route to the word or words involved. It does not need to refer to the AOT, and in fact need not examine the capability stored in the window register.

A virtual address (capability, offset) does not provide a convenient hardware route to a word. Since a word may be specified by more than one virtual address, hardware recognition of virtual addresses would be complicated. It would be easy to associate with each window register the information needed to convert offsets relative to the window into offsets relative to the start of the containing object, thereby yielding a unique address for any word. However, the resulting object name, offset pair would be very large. Some reduction could be achieved by assigning a 'segment number' to every object of current interest as a temporary but reasonably long-lived synonym for the object name. We have chosen to use a logically similar but more compact scheme.

In our implementation the 'hardware route' component of a view is a base address in a linear Intermediate Address Space (IAS). The IAS is much larger than the physical main memory, but small enough to allow a manageable address width (32 bits in our present system, but the width is not fixed by the architecture). Every object of current interest is mapped into this space as a single block of consecutive words. This mapping is recorded in the AOT, which contains entries for all objects currently mapped into IAS.

When a capability is loaded into a window register, the IAS base address corresponding to the beginning of the window, the window size and access rights are retained by the processor in an extension of the window register. A logical address (window register number, offset) is converted by the processor to an IAS address by adding the offset to this base (Fig. 1). Checks are also performed on the access rights and size. The IAS is large enough to allow an object to retain its IAS location for some hours. Hence the view information associated with a window register is not very volatile, and can usefully be retained in a process when it is removed from a processor for a short period, e.g. to fix a page fault.

IAS addresses produced by processors are recognised and translated to physical addresses by fast hash tables in the memory modules of the multiprocessor. By having the processors generate and the memory modules recognise IAS addresses rather than physical memory locations, movement of a page of an object between a volume and physical memory does not require adjustment of any view of the object held in a window register.

The use of retained views in the window registers of a process allows efficient references to objects by bypassing the full mechanics of the virtual-memory interface, but

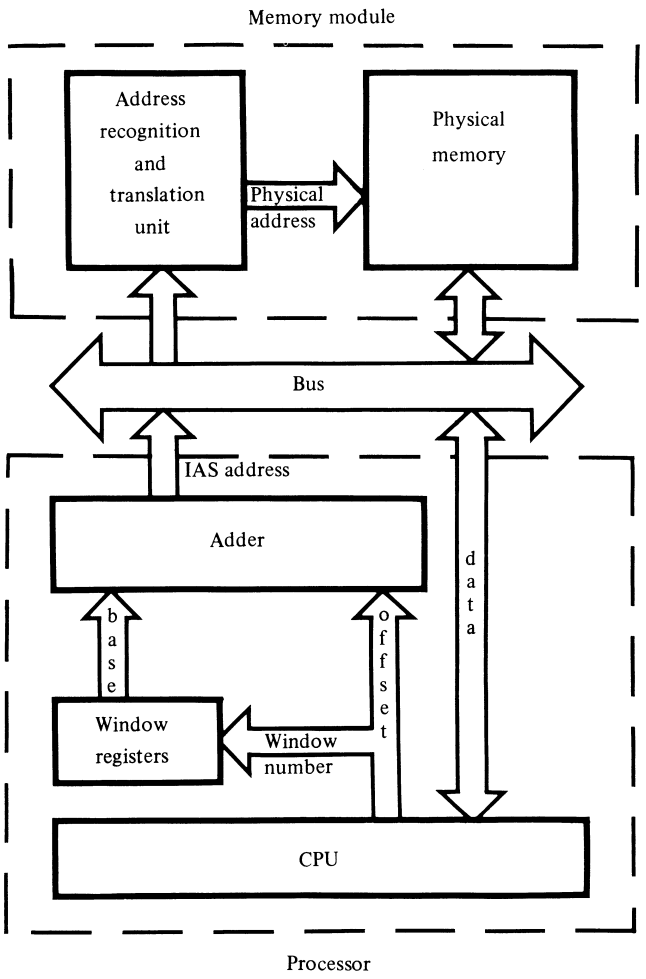

Figure 1. Hardware Addressing

raises the possibility that a change to the status of a capability may not be noticed by a process which has that capability in a window register. The only change of status that needs to be considered is the destruction of the capability. If the destruction is a consequence of the destruction of the master capability, i.e. of the object, no problem arises since all pages of the object are removed from physical memory and the memory modules will cease to recognise the IAS addresses lately occupied by the object. However, if the capability destroyed is a derived one, other steps must be taken to invalidate any window register view associated with the capability.

We do not attempt to locate and destroy all such views which may be held in processes, but provide two means for invalidating them. First, no process relies on the continued validity of any view which it has not checked in the last few seconds. The next use of a window register containing such a view will cause the process to check that the capability still exists. If it does, the view-defining information in the window register is assumed still to be valid. Thus, when a derived capability is destroyed, the destruction is guaranteed to be effected within a fixed but not negligible time. We expect this slow destruction will not suffice in most cases.

A prompt destruction is also provided, but is more expensive. It is effected by relocating the object in IAS, which can be done without moving any of its pages in physical memory. Attempts to use old views of the object then result in a pseudo-page fault. Processes with valid capabilities for the object can resolve the fault and recover the new IAS address of the object, but users of the destroyed capability will fail. At infrequent intervals, processes are forced to revalidate the IAS addresses in their window register views, to allow re-use of vacated areas of IAS.

The periodic checking on the validity of a view in a window register is no worse than checking undertaken in other systems with hardware support for capabilities. For example, the CAP[18] flushes its implicitly loaded cache whenever a capability is copied or refined, hence forcing a process to re-evaluate the validity of capabilities it uses.

11. CONCLUSION

Password-checked capabilities, like tagged capabilities, may be freely mixed with other data and need not be segregated. Unlike tagged capabilities, they are ordinary data values, and so can be held outside the computer system. They can be protected by encryption when passed through an untrusted agent or insecure medium. A simple form of encryption can be used to implement a confinement mechanism.

As in some segregated-capability schemes, capabilities may readily be revoked without a time-consuming search for instances. The dependence tree of derived capabilities provides secure revocation without needing any restriction on copying capabilities or making derivatives.

Although the use of a capability conceptually requires an indirection to recover access and location information, the view mechanism outlined provides an efficient implementation so long as window registers are not reloaded too often. The architecture thus favours use of capabilities for relatively large and stable objects, but the overheads incurred in a more volatile use of capabilities are not intolerable.

The money system is an important feature of the virtual memory. It allows recovery of garbage even though accessibility of objects cannot be determined, removes resource allocation deadlocks from the concern of the system, and provides a flexible accounting mechanism which, for instance, can require processes to pay for the use of proprietary packages.

The design of the capability mechanism requires no concept of ownership, either for object management or accounting, and imposes no artificial hierarchic structure on the virtual memory. It is 'open' in the sense that objects, including processes, need not be subjected to any authorisation procedure before becoming usable or active. A communication line from another computer or from a terminal can be given direct access to the virtual-memory interface, and hence can create objects, talk to processes, read and alter objects or anything else for which it can show a capability.

Acknowledgements

The design of the virtual memory architecture owes much to the suggestions and criticisms of David Koch, Ken McDonell, David Abramson and John Rosenberg.

M. Anderson and R. D. Pose gratefully acknowledge the financial assistance provided by a Commonwealth Postgraduate Research Award and a Monash Graduate Scholarship respectively.

The multiprocessor project is supported by the Australian Research Grants Scheme (no. F 8315169 I) and a Monash Special Research Grant.

REFERENCES

1. D. Abramson, Computer hardware to support capability based addressing in a large virtual memory. *Ph.D. thesis*, Monash University (1982).
2. E. Cohen and D. Jefferson, Protection in the Hydra operating system. *Proceedings of the Fifth Symposium on Operating System Principles* (1975).
3. R. S. Fabry, Capability based addressing. *Communications of the ACM* **17** (7) (1974).
4. R. J. Feiertag and P. G. Neumann, The foundations of a probably secure operating system. *AFIPS Conference Proceedings, NCC 79*. (1979).
5. E. F. Gehringer, Variable length capabilities as a solution to the small object problem. *Proceedings of the Seventh Symposium on Operating System Principles* (1979).
6. C. G. Girling. Object representation on a heterogeneous network. *Operating Systems Review* **16** (4) (1982).
7. A. K. Jones, R. J. Chansler, I. D. Durham, K. Schwans and S. R. Vegdahl, STAROS, a multiprocessor operating system for the support of task forces. *Proceedings of the Seventh Symposium on Operating System Principles* (1979).
8. J. Keedy, Support for Software Engineering in the Monads Computer Architecture. *Ph.D thesis*, Monash University (1982).
9. B. W. Lampson, A note on the confinement problem. *Communications of the ACM* **16** (10) (1973).
10. B. W. Lampson and H. E. Sturgis, Reflections on an operating system design. *Communications of the ACM* **19** (5) (1976).
11. C. E. Landwehr, C. L Heitmeyer and J. McLean, A security model for military message systems. *Transactions on Computer Systems* **2** (3) (1984).
12. S. B. Lipner, A comment on the confinement problem. *Proceedings of the Fifth Symposium on Operating System Principles* (1975).
13. G. J. Myers and B. R. S. Buckingham, A hardware implementation of capability based addressing. *Operating Systems Review* **14** (4) (1980).
14. F. J. Pollack, K. C. Kahn and R. M. Wilkinson. The iMAX432 object filing system. *Proceedings of the Eighth Symposium on Operating Systems Principles* (1981).
15. A. S. Tanenbaum and S. Mullender, An overview of the AMOEBA distributed operating system. *Operating Systems Review* **15** (3) (1981).
16. M. W. Wilkes, Hardware support for memory protection: capability implementations. *Proceedings of a Symposium on Architectural Support for Programming Languages and Operating Systems* (1982).
17. M. V. Wilkes, Security management and protection – A personal approach. *The Computer Journal* **27** (1) (1984).
18. M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its Operating System*. Elsevier North-Holland, (1979).
19. W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack, HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM* **17** (6) (1974).
20. W. A. Wulf, R. Levin and S. P. Harbison, *HYDRA/C.mmp. An Experimental Computer System*. McGraw-Hill, London (1981).