Supporting large persistent stores using conventional hardware†

Francis Vaughan, Alan Dearle
Department of Computer Science, University of Adelaide
Adelaide, Australia

Abstract

Persistent programming systems are generally supported by an object store, a conceptually infinite object repository. Objects in such a repository cannot be directly accessed by user programs; to be manipulated they must be fetched from the object store into virtual memory. Thus in these systems, two different kinds of object addresses may exist: those in the object store and those in virtual memory. The action of changing object store addresses into virtual memory addresses has become known as *pointer swizzling* and is the subject of this paper.

The paper investigates three approaches to pointer swizzling: a typical software address translation scheme, a technique for performing swizzling at page fault time and finally a new hybrid scheme which performs swizzling in two phases. The hybrid scheme supports arbitrarily large pointers and object repositories using conventional hardware. The paper concludes with a comparison of these approaches.

1. Introduction

Most persistent and database programming languages are supported by an object store, a conceptually infinite repository in which objects reside. In order to manipulate these objects, they must be fetched from the object store into directly addressable memory, usually virtual memory. In systems which support orthogonal persistence [3] this is performed transparently. Thus in these systems, two different kinds of object addresses may exist: those in the backing store (persistent identifiers or PIDs) and those in directly addressable memory (virtual addresses).

Many researchers have argued that large pointers (anywhere up to 128 bits) are required to support persistent systems [9, 17]. Persistent pointers need not be the same size as those supported by virtual memory (usually 32 bits); indeed persistent identifiers may be arbitrarily long. This paper presents a new architecture which supports arbitrarily large pointers and persistent stores using conventional hardware.

The persistent address of an object may be mapped onto a virtual address in a number of ways:

- Dynamically translate from a PID to a virtual address on each dereference.
- Make an object's virtual address coincident with its persistent identifier.
- Perform a once only translation from a persistent identifier to virtual address, overwriting the copy of the persistent identifier in the virtual address space with a virtual address so that all subsequent dereferences incur no translation penalty.

This last option has become known as *pointer swizzling* and is the subject of this paper. The first option, dynamic translation, is seldom more efficient than swizzling [16]. The second option is only possible if persistent stores are small enough to be contained within

† In Proceedings of the 5th International Workshop on Persistent Object Systems, San Miniato, Italy, September 1992, to appear in Springer-Verlag Workshops in Computing Series.

the virtual memory. All these techniques have been used to implement persistent object stores [8, 12, 13].

Pointer swizzling may be performed at a variety of times, the earliest being when objects are loaded or faulted into memory; this is termed *eager pointer swizzling*. The latest time swizzling may be performed is when a pointer is dereferenced, and is termed *lazy pointer swizzling*. When swizzled objects are removed from virtual memory, virtual memory pointers must be replaced by PIDs; this is often referred to as *unswizzling* or *deswizzling*.

Eager pointer swizzling has some advantages; in particular, if a data set may be identified in its entirety, all the pointers may be swizzled at once, avoiding the necessity to test whether a reference is a PID or a virtual address prior to every dereference. However, this approach has the disadvantage that pointers may be swizzled, involving some computational expense, and never used.

Some systems use an *ad hoc* swizzling scheme; in these systems persistent pointers are the same size as VM addresses and may be coincident with the virtual address space. Whenever possible data is simply copied at the appropriate position into the virtual address space from the store. However if the appropriate region has already been allocated, swizzling is employed. Such systems are unable to support persistent stores larger than virtual memory and are not discussed further. It is believed that a variation of this scheme is also used by Object Design [15] .

In persistent systems it is unusual to be able to identify a self contained data set and some lazy swizzling is unavoidable. The first persistent systems to employ swizzling [4] relied upon a software test to distinguish between PIDs and local addresses. Recently, schemes have been described which avoid performing these tests by performing pointer swizzling at page fault time [18]. In this paper we present a new hybrid technique which offers many of the advantages of both these approaches.

The remainder of the paper is structured as follows: firstly we will describe a typical software address translation scheme. This is followed by a discussion of Wilson's scheme: a technique for performing pointer swizzling at page fault time. Next we introduce a new scheme which is a hybrid and performs swizzling in two phases and an analysis of this scheme is made. We also suggest some implementation techniques that may be utilised in conjunction with such a scheme. The paper concludes with a comparison of the three architectures.

2. Software address translation

The first object systems to be called persistent [4, 5] performed lazy pointer swizzling implemented entirely in software. In this section, for illustration purposes, we will concentrate on one of these, the Persistent Object Management System written in C, the CPOMS [7]. The CPOMS is the underlying system used to support implementations of PS-algol [2] under Unix.

The persistent store implemented by the CPOMS is a large heap with objects being addressed using persistent identifiers (PIDs). How PIDs are interpreted is not relevant to this paper and the interested reader is referred to [6] for more details. PIDs may be arbitrarily large but in current implementations PIDs are identical in size to the normal pointers (known as Local Object Numbers or LONs) used by the PS-algol run time system [1] . PIDs are distinguished by having their most significant bit set. Hence it is possible for the PS-algol run time system to distinguish between a LON and a PID.

PIDs are pointers to objects outside of the program's virtual address space, therefore the objects to which they refer cannot be directly addressed by a PS-algol program. To

ensure that PIDs are not dereferenced, a test is made prior to the use of any object address; in the PS-algol system this test is made using in line code. When an attempt to dereference a PID is detected, the referenced object is fetched into memory and the PID is swizzled and replaced with the appropriate LON. This process is shown in Figure 1 below in which objects B, C and E have been fetched into directly addressable memory where they are represented by objects B', C' and E'. Note that some references within virtual memory are virtual memory addresses whereas other are PIDs.

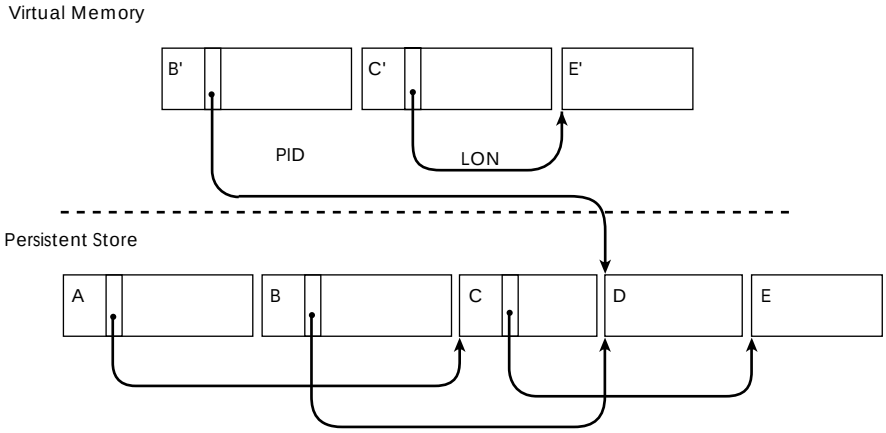

Figure 1: Swizzling in PS-algol

In order to prevent more than one copy of an object being made, a data structure called the PID to Local Address Map (PIDLAM) is kept. When a PID is first used and the object to which it refers is copied into local memory, the PID is entered into the PIDLAM along with the LON of the copy as shown in Figures 2 and 3. Therefore, if another instance of the same PID is encountered, the LON of the copy can be found from the PIDLAM. This is necessary to preserve referential integrity in the running system.

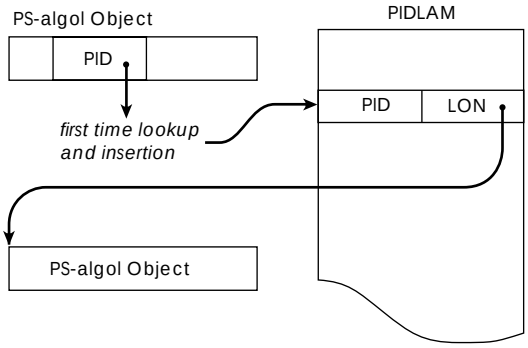

Figure 2: Looking up a PID in the PIDLAM

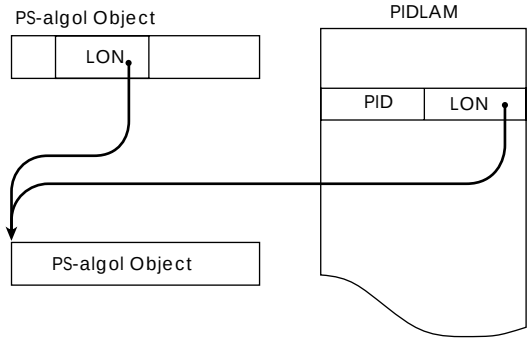

Figure 3: Overwriting a PID by a LON

Although relatively simple, this mechanism compromises performance in five areas:

- all the address translation is performed in software,
- all pointer dereferences must be checked using software to ensure that the pointer is not a PID,
- disk fetches occur on a per object basis,
- large objects must be copied into virtual memory in their entirety, and

- every unswizzled pointer to an object must be swizzled at the time of dereference, even if the referend is resident in local memory.

The first, fourth and last of these problems may be eliminated if the hardware address translation mechanisms may be exploited. As stated earlier, this is only possible if the persistent identifier of an object is made coincident with its virtual address; clearly this approach may only be used with relatively small stores. The second problem may be eliminated if persistent addresses are illegal virtual memory addresses since an access will cause the hardware to raise an exception. This is only more efficient if the operating system provides a light weight exception mechanism. The CPOMS partially addresses this problem by eagerly swizzling certain pointers and in so doing avoids some checks. For example, pointers loaded onto a stack in the dynamic call chain are eagerly swizzled. The third problem may be overcome by amortising the cost of disk access across many object fetches.

3. Address translation at page fault time

Recently, an approach has been suggested by Wilson [18] that employs both pointer swizzling and page faulting techniques. The basic strategy is to fetch pages of data into virtual memory rather than individual objects. As pages are fetched, they are scanned and all (persistent) pointers are translated into valid virtual memory addresses. References to non-resident objects cause virtual memory to be allocated; these pages are fetched only if the pointers into them are dereferenced. In Wilson's scheme, pages of data in virtual memory only contain valid virtual memory addresses, never persistent identifiers.

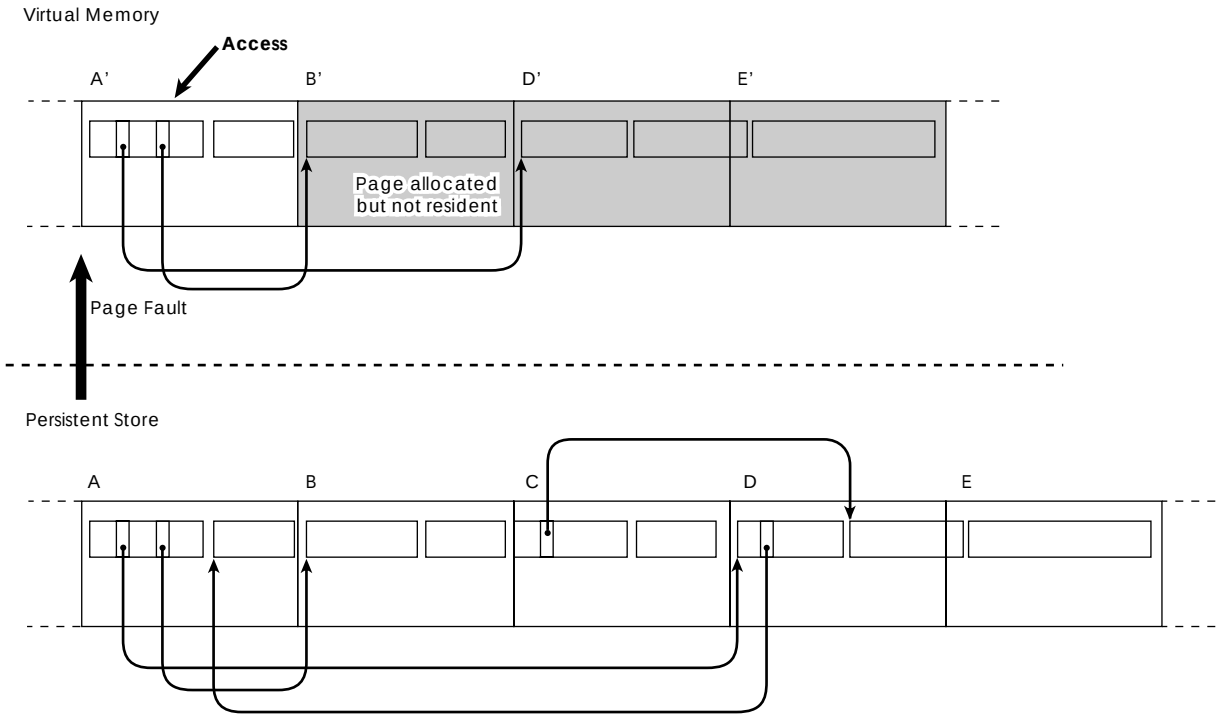

Figure 4: Page faulting and allocation in Wilson's scheme

Figure 4 shows Wilson's scheme in operation; in the diagram, a non-resident persistent object on page A (i.e. an object on a page that has not been fetched into virtual memory)

has been accessed. This will cause a copy of page A, denoted A', to be fetched into virtual memory. At this time, the page is scanned and all the pointers in it are swizzled into valid virtual memory addresses. Since page A contains references to objects on pages B and D, locations for pages B' and D' must be allocated in virtual memory and the pointers into those pages swizzled to the addresses of B' and D' with appropriate offsets added. Virtual memory must also be allocated for page E since objects from page D overlap that page. Note that the loading and swizzling of pages B', D' and E' is performed lazily: only space is allocated for them in virtual memory. This mechanism causes virtual memory which may never be used to be allocated. Since pages B, D and E may have already been faulted into virtual memory, a translation table similar to the CPOMS PIDLAM must be maintained to avoid loss of referential integrity.

When a reference to a previously unseen page is encountered whilst scanning an incoming page, three actions are required. Firstly a new translation table entry for the page is allocated. Secondly, the store is interrogated to discover the page's crossing map (described below). Thirdly, virtual memory space is allocated for the page. Interrogation of the store is potentially expensive and since it is performed eagerly, at page fault time, is a potential performance bottleneck.

When a page is scanned, it is necessary to find all the pointers on that page; provided that objects are self describing, this requirement reduces to finding the header of the first object on or overlapping the page boundary. This same requirement is made of object systems by some parallel garbage collection techniques [11, 14] and the solutions are well known. The first solution is to maintain a bitmap known as a *crossing map* which indicates if an object header is coincident with the start of a page.

To find all the pointers on a page, the system has to scan the pages which precede the faulted page starting at the first page which has a object header coincident with the start of the page. This obviously can be expensive if there is a high degree of crossing and the pages are mostly on disk. Another technique is to maintain an array of pointers containing one pointer for each page in the system. Each pointer points to the first object header before or aligned with the start of the page. In this way, at most two pages need to be examined when a page is faulted.

If pointers are stored contiguously in objects a further optimisation is possible. Rather than an array of pointers, an array of tags is maintained, with each tag corresponding to a page in the store. Each tag, which may be encoded into 32 bits, describes any partial object which may overlap the start of the page. The tag consists of the length of the partial object (if any), the offset of the first pointer in the partial object (if any) and the number of pointers in the partial object (if any). This optimisation means that only the faulted page needs to be examined when a page fault occurs.

In Wilson's scheme, page evacuation from virtual memory is convoluted. This problem is exacerbated by the fact that virtual memory is eagerly allocated and hence the need to reuse virtual memory addresses potentially more frequent. If a set of pages is written back to persistent storage, the pointers in those pages must be deswizzled into PIDs by consulting the translation table. However, if virtual memory is exhausted and a virtual memory range is to be reused by another persistent page, all pointers which refer to the old contents must be removed.

A translation table that contains an entry for each instance of a referend object can become very large. Wilson proposes a scheme in which the translation table provides a per page rather than per object mapping. To implement this, PIDs are structured so that the offset within the holding page of an object is encoded into the object’s PID. For example, assuming 8k byte pages and word alignment of objects, eleven bits are needed

to describe the offset. This leaves 53 bits of a 64 bit PID to identify the page. The structure of PIDs is depicted in Figure 6.

This scheme has two advantages. First, it is only necessary to maintain a mapping from pages within the large persistent address space to pages in the machine virtual address space. This table is relatively small and of fixed size. Secondly, an object's offset is required in the construction of a swizzled pointer. If the offset were not coded into the PID, further interrogation of the store manager would be required, adding extra cost to the swizzling process.

4. A hybrid approach

The CPOMS and systems like it require software tests prior to each object dereference to check if the pointer being dereferenced is a persistent identifier. Wilson suggests that pointer swizzling may be performed at page fault time. This implements a barrier that ensures that a running program may never encounter a PID. However this is not achieved without cost; space must be allocated in virtual memory for every page referred to by data resident in virtual memory. Whilst this does not seem too onerous it has some unfortunate consequences.

Firstly, space in virtual memory is allocated greedily, this may cause virtual memory to become exhausted even although much of it has not been used. The counter argument says that many programs will have a high degree of locality of reference. However consider an array of large objects such as images – whenever the array is faulted into memory, enough virtual memory must be allocated for all the referenced images. It is likely that such an operation would be common in persistent applications although uncommon in traditional database applications.

We now present a hybrid architecture which does not require software checks for pointer validity and does not involve greedy allocation of virtual memory. The architecture is designed to support PIDs which address a space much larger than virtual memory and makes the requirement that PIDs are at least twice as large as virtual memory addresses. From this point on, to ease discussion, we will assume that a PID is 64 bits and virtual memory pointers are 32 bits.

In this architecture, pointers are swizzled in a two phase process: first at page load time to refer to an entry in a translation table and secondly to a virtual address when the referend object is first accessed. When pages are first accessed, they are copied from persistent memory into the virtual address space and scanned to find the pointers contained in them. Rather than allocating virtual memory for every page referenced by the page being faulted in, as happens in Wilson's scheme, the long pointers contained in the page are swizzled to refer to either:

- entries in a translation table if the referend object is not present in virtual memory (*partially swizzled*), or
- a virtual memory pointer (*fully swizzled*) if it is.

This is shown in Figure 5 below.

The translation table used in this scheme may be similar to either the one used by the CPOMS (a per object translation table) or by Wilson (a per page table). The table contains the persistent and virtual address (if any) of all objects (or pages) referred to by objects resident in virtual memory. For the remaining discussion we will assume a per page translation table. Unlike the CPOMS, the table is protected from any access by the user process, thus when a partially swizzled pointer is dereferenced an access fault occurs.

This triggers the second phase of the swizzle in which the pointer (currently containing the table entry address) is overwritten with the virtual address of the referend.

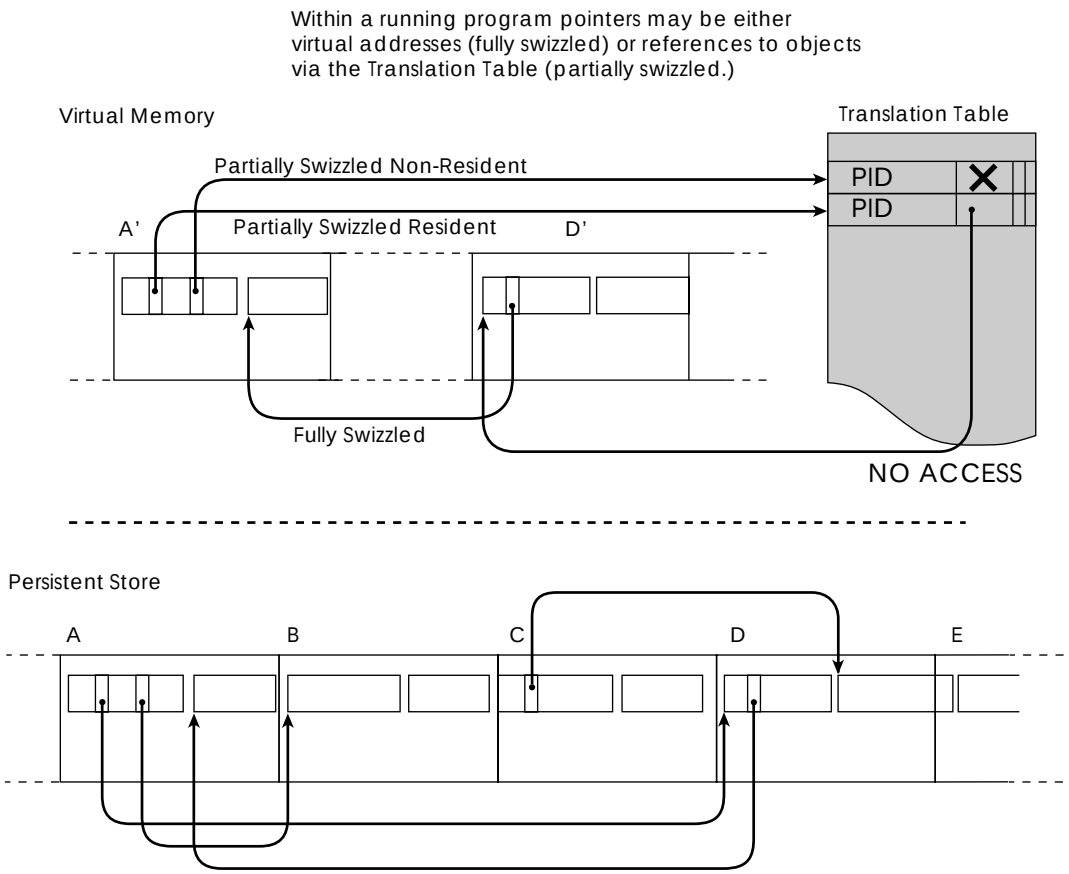

Figure 5: Partially and fully swizzled pointers

If the referend is not resident in virtual memory, the page containing it must be loaded from the persistent store. To do this, the PID, which may be found in the translation table, must be presented to the store manager. Using this the store manager can supply the appropriate page(s) containing the object. Once the page is loaded the partially swizzled pointer is overwritten with the virtual address of the object and the object dereference can proceed. The page load may result in new entries being created in the translation table. In contrast to Wilson's scheme it is only when an object is used that the store is interrogated to discover how much virtual memory must be allocated.

When a persistent pointer is fully swizzled half the space in the pointer is unused – this space is used to store the address of the corresponding translation table entry. This allows the pointer to be easily deswizzled.

In a partially swizzled pointer the space is used to store the offset within the page at which the object begins. This offset, when combined with the address at which the page is placed when it is faulted into virtual memory, forms the object address of a fully swizzled pointer. The store formats for pointers and the translation table entries are shown in Figure 6.

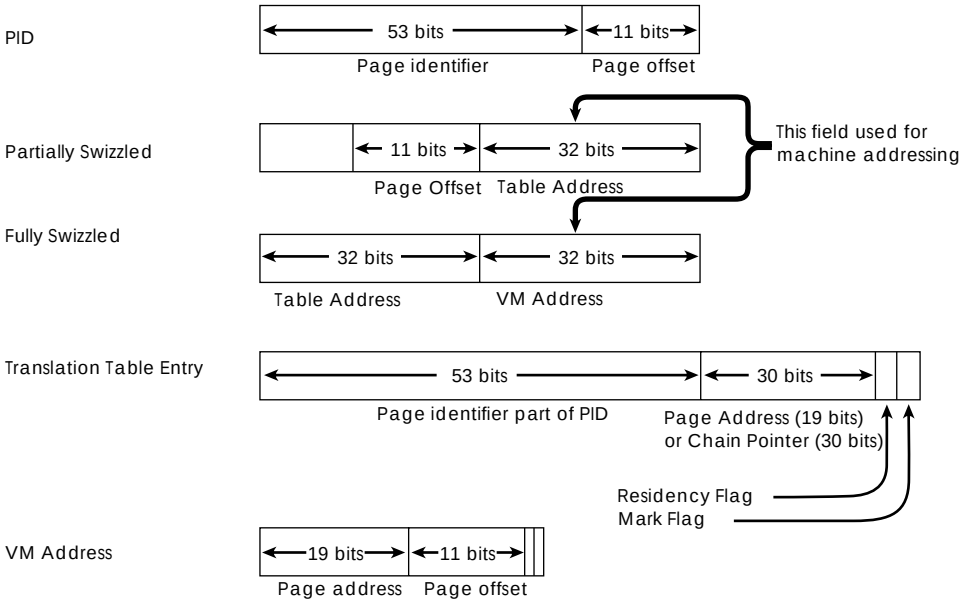

Figure 6: Pointer and translation table formats

The translation table maps from page identifiers in the persistent store to pages within the machines virtual address space. Each translation table entry holds the page identifier field of a persistent identifier, a virtual memory address, a residency bit and a mark bit. If the residency bit is set the virtual memory address holds the address of the corresponding page in memory, otherwise it may contains the head of a partially swizzled pointer chain which is discussed next. The formats depicted in Figure 6 assume a 32 bit virtual address space and a page size of eight kilobytes.

4.1. Eager Swizzling

The eager swizzling technique described by Wilson has the advantage that when a page is faulted into memory all the pointers which refer to objects on that page are automatically correct (since those pointers already refer to the correct virtual addresses on that page). A late swizzling scheme does not have this advantage, however this may be simulated. A form of eager swizzling can be provided by threading a linked list called the *partially swizzled pointer chain* through of all instances of pointers referencing objects on a page. When an object is faulted into memory the swizzling code not only swizzles the pointer that caused the fault, but follows the chain and swizzles as many other pointers as it can. This is eager pointer swizzling; as discussed earlier, this is only more efficient if some of these pointers are used. This very much depends on the nature of the system, programs and programming languages being used and the marginal costs of creating and following the pointer chains versus the cost of on demand per pointer swizzling.

As described the pointer formats do not provide space for the link field needed to implement the partially swizzled pointer chain. The chain may be implemented by using one of the following:

- Making PIDs large enough to accommodate the link. Expanding PIDs to 96 bits also has the advantage of providing a much larger address space.
- Using a per object translation table. Using this technique the translation table pointer field in a partially swizzled pointer uniquely describes the referend object. The upper half of the pointer does not contain the page

offset and is free to hold the link field. However per object translation tables can become very large.

- By encoding the information. The problem is that 30 bits are required to implement the chain (assuming word alignment.) The table address requires 28 bits (assuming 16 byte table entries), the offset requires 11 bits, leaving only 25 bits free. Therefore another five bits are required. These bits may be stolen from the table address if the translation table is made 32 times as large as normally required.

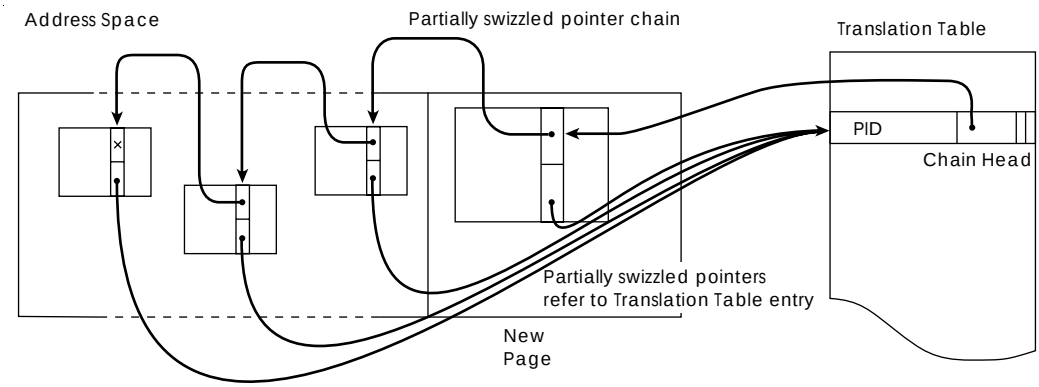

Figure 7: A pointer is inserted into the partially swizzled pointer chain

The partially swizzled pointer chain is formed as pages are loaded into virtual memory. If an instance of a PID is encountered which is already in the translation table, the head of the partially swizzled pointer chain is loaded into the unused space in the partially swizzled pointer and the address of the new instance is copied into the chain pointer head stored in the translation table entry. This process is shown in Figure 7 above.

During the execution of a program, some of the pointers in the partially swizzled pointer chain may have been overwritten by the user making (64 bit) pointer assignments. Such a break is simple to detect when the chain is being scanned since an overwritten pointer will not refer to the expected table entry. If the chains are broken, it is not possible to find all the instances of a partially swizzled pointer. However, the remains of the chain will continue to exist and many of the pointers in it may be still be swizzled through the partial chains referenced by the translation table entry and the pointer being swizzled. Also, future dereferences of pointers in a partial chain will permit yet more pointers to be found and swizzled at low cost. It is possible to maintain intact pointer chains by requiring that code doing pointer assignments perform list insertion and deletion as part of the assignment process. We consider that this would be too expensive for the marginal gains.

4.2. Deswizzling

Virtual memory addresses may only be interpreted inside the address space in which they were created. Therefore the only meaningful addresses that can be used in pages outside of a virtual address space are PIDs. The necessity to make copies of pages outside of a virtual address space arises for two reasons:

- to send pages to a process resident within another virtual address space,
- to send pages back to the persistent store.

This requires the pointers within the page copies to be fully deswizzled (PIDs). This is performed by following the reference to the translation table entry contained within the pointer and overwriting the pointer with the PID found in the table.
The management of pages within the virtual address space involves the allocation and control of two resources:

- physical memory, and
- virtual memory.

Physical memory is a finite resource and will rarely be large enough to hold the working set of pages used by a program. Pages will be removed from physical memory either to make room for another page needed for computation to continue, or when data is shared between separate virtual address spaces. When a page is removed from physical memory, pointers within it must be deswizzled as described above. A page which is not resident in physical memory may still reside within the virtual address space of the process.

In a persistent operating system [10] the integration of swap space and persistent storage may give considerable advantages. We will therefore assume that pages removed from physical memory are either returned to the persistent store or to another persistent application.

Virtual memory is also a finite resource. Programs that use very large data sets or those which are very long lived may eventually exhaust virtual memory. Indeed, the architecture described in this paper is designed to support such programs. When virtual memory is exhausted, virtual address ranges require reuse in a manner analogous to the reuse of physical memory. It should be noted that both Wilson's scheme and the hybrid design require that virtual memory addresses be reallocated in such a way that the reallocated ranges do not divide objects.

When a page is removed from the virtual address space, it must also be removed from physical memory if resident. At this time all references to that page from within virtual memory must also be removed. This involves ensuring that all references to objects in the removed page are partially swizzled pointers by deswizzling the appropriate fully swizzled pointers.

4.2.1 Deswizzling in Wilson's Scheme

Wilson proposes a scheme to reclaim pages of virtual memory that works as follows. Initially all of virtual memory is protected from access. Whenever the mutator attempts to access a page that is protected from access two actions are taken. First, the page protection is removed. Next, the page is scanned to find all pointers on it and any referenced pages are marked. Finally the mutator is resumed. As the mutator executes it constructs a new working set of pages. At some time in the future any page that is neither open for access nor marked as referenced may be reused. Once page reuse has begun it is possible that when a protected page is scanned a pointer to a reused page will be encountered. When this occurs a new range of virtual addresses must be allocated and the pointer changed to refer to this new location. This process is similar to the greedy allocation that occurs when a page is retrieved from the persistent store described earlier.

4.2.2 Deswizzling in the Hybrid Scheme

The hybrid scheme provides greater flexibility in address space reuse. Since partially swizzled pointers do not directly reference virtual addresses, fully swizzled pointers may be replaced with partially swizzled. This allows address ranges within virtual memory to

be reused whilst references to objects that once resided within those addresses remain in virtual memory. In the hybrid scheme page reuse occurs as follows.

During normal execution a candidate set of page ranges can be identified for reuse, using conventional LRU techniques. This may be integrated with the LRU scan used to manage allocation and reuse of physical memory. When it becomes necessary to reuse virtual address ranges, access to virtual memory is denied as in Wilson's scheme. However, in the hybrid scheme reuse can proceed immediately. Those address ranges considered as candidates for reuse may be reused as soon as their contents are secure in the stable store. An exception will occur on the first access to a page since reuse started, again the exception handler scans the page in the same manner as Wilson's scheme. However rather than allocating new address ranges for those pointers that reference reused addresses, pointers to objects within reused address ranges may be replaced with their partially swizzled form. Thus partially swizzled pointers serve two purposes: to permit virtual memory to be deallocated at low cost and as a mechanism to avoid greedy allocation of virtual memory.

In addition to the mutator causing pointers on pages to be deswizzled, it is advantageous to provide a parallel sweep of virtual memory that eagerly scans pages and deswizzles pointers. Once all virtual memory has been swept, all allocated pages will be open for access and no direct references to deallocated pages will exist. The mutator can attempt to reference a page that is tagged for reuse by dereferencing through a partially swizzled pointer. If this page has not been reused and is still resident in memory it need only be removed from the reuse set and scanned for pointers. The partially swizzled pointer is fully swizzled and execution continues. It is not necessary to reuse all address ranges tagged for reuse. At any time ranges can be removed from the reuse set and references to objects within them left intact.

The ability to choose the number of pages to be reused ahead of time, which pointers to deswizzle, and the rate of progress of the parallel sweep provide useful tuning parameters to the memory management system. Setting the system to label all pages as reused, and to untag any referenced pages upon page scan effectively reduces to Wilson's scheme. Labelling all pages as reused, and deswizzling all pointers encountered effectively frees the entire virtual address space. A complete spectrum of choices is available within these extremes.

4.3. Elaboration of detail

The above description glosses over a large number of important details namely:

- finding object addresses,
- pointer comparisons,
- large objects,
- management of the translation table,
- creation of new objects,
- exception handlers, and
- access to the translation table.

We will now proceed to describe these implementation details.

4.3.1. Finding object addresses

When an access is attempted through a partially swizzled pointer three actions are required:

1. find the object to which access is being attempted,
2. overwrite the pointer with the virtual address of the referend, and finally,
3. update the saved state of the executing code's register set to refer to the object.

None of these activities is straightforward, and requires detailed study at the basic level of the machine's operation. Consider the code fragment shown in Figure 8 below, a type *tuple* is declared to be a record and an instance of that type is created. Later in the program a field of an instance of type *tuple* is dereferenced.

```
type tuple is record( a,b,c,e,f,g : integer )
let an_instance := tuple( 1,2,3,4,5,6 )
.....
write an_instance.f
```

Figure 8: Dereferencing a field of a record.

Consider the implementation of the program above. The pointer denoted by *an_instance* may be partially or fully swizzled; an aim of the architecture is to avoid user code having to test which of these it is. Fully swizzled pointers do not present a problem: the dereference is performed without incident. A partially swizzled pointer will result in an attempt to access an address within the translation table and this will cause an access fault. However, the address that causes the fault will not be the address of *an_instance*'s translation table entry since an offset will have been added to the object pointer in order to extract the field. Hence, although an access fault will deliver the address of the fault to the exception handler, the address will not directly resolve the identity of the required object. Similar problems occur in the other two phases; the swizzling code must be able to find and swizzle the object pointer, but ordinarily there is no record of the location of that pointer. If this swizzle is not performed, the system reduces to a translation per dereference design.

In the hybrid system, the saved state of the executing thread is repaired by an exception handler which must therefore be able to determine which machine registers contain the addresses requiring change. This can be arbitrarily difficult; to make the problem tractable steps must be taken to ensure that when an object reference is made, it must be performed in such a way that allows the recovery of the information needed to complete the swizzle. This requires a specification of the object access process at the machine code level.

All of the information required will ordinarily pass through the processor during the execution of a dereference sequence. The difficulty is in keeping track of this information and making it available to the exception handler. A similar sequence is executed whether the access is a read or a write. In general a dereference takes place in three steps and is shown in Figure 9 below:

1. The address of the pointer to the head of the object being referenced is loaded into a register.
2. Using that address, the address of the object is loaded into a register.

3. The offset within the object is added to the object address and the result used as the address of the memory access.

For the mechanism described in this paper to work, the only changes required to this sequence are to ensure that the pointer address is not overwritten after the pointer value is loaded (which ordinarily is a legal optimisation) and to ensure that the instruction sequence always uses the same registers for this purpose, allowing the exception handler to find the necessary addresses.

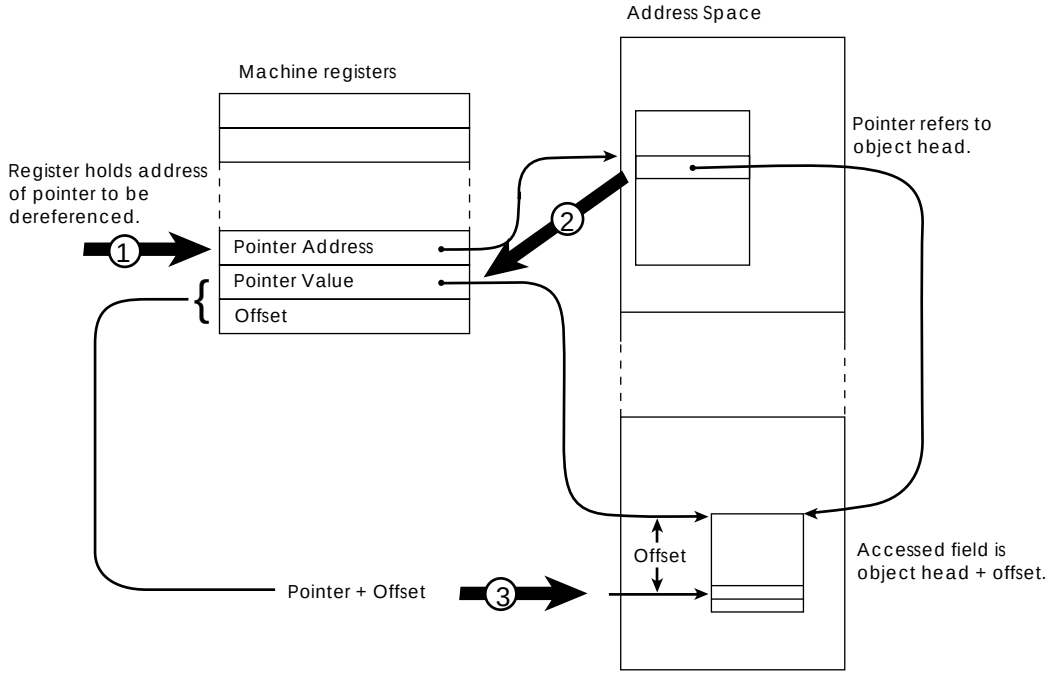

Figure 9: The three steps in pointer dereference

The result of these restrictions is a scheme in which during a dereference operation two registers are reserved for particular purposes. Firstly a *Pointer Pointer* register is loaded with the address of the pointer to the object being dereferenced. Next the *Object Pointer* register is loaded with the value referenced by the Pointer Pointer register. This value is either the address of the head of the object (for fully swizzled pointers) or the address of a translation table entry (for partially swizzled pointers). Finally, the offset is added to the contents of the Object Pointer register (with a single indexed addressing mode instruction) and the result used as an address to effect the dereference. If the pointer is partially swizzled an access exception will occur. The exception handler will receive an address within the translation table, allowing it to distinguish the exception from any others that may occur. In processing the exception the exception handler places the fully swizzled value of the pointer in both the location referred to by the Pointer Pointer register and into the Object Pointer register, then the instruction that caused the exception is restarted. If the pointer is fully swizzled then the instruction will execute without incident and with no extra cost. This process is shown in Figures 10 and 11 below. The Pointer Pointer and Object Pointer registers are only special during the process of a dereference, they are available for general use at other times.

The mechanism relies on the translation table residing in protected memory and an exception being raised when access to that memory is attempted. When the offset is added to the Object Pointer it is possible for a legal memory address to be generated. This

may be avoided if the translation table is positioned in high memory and grows downward and that the exception mechanism checks for arithmetic overflow during the addition.

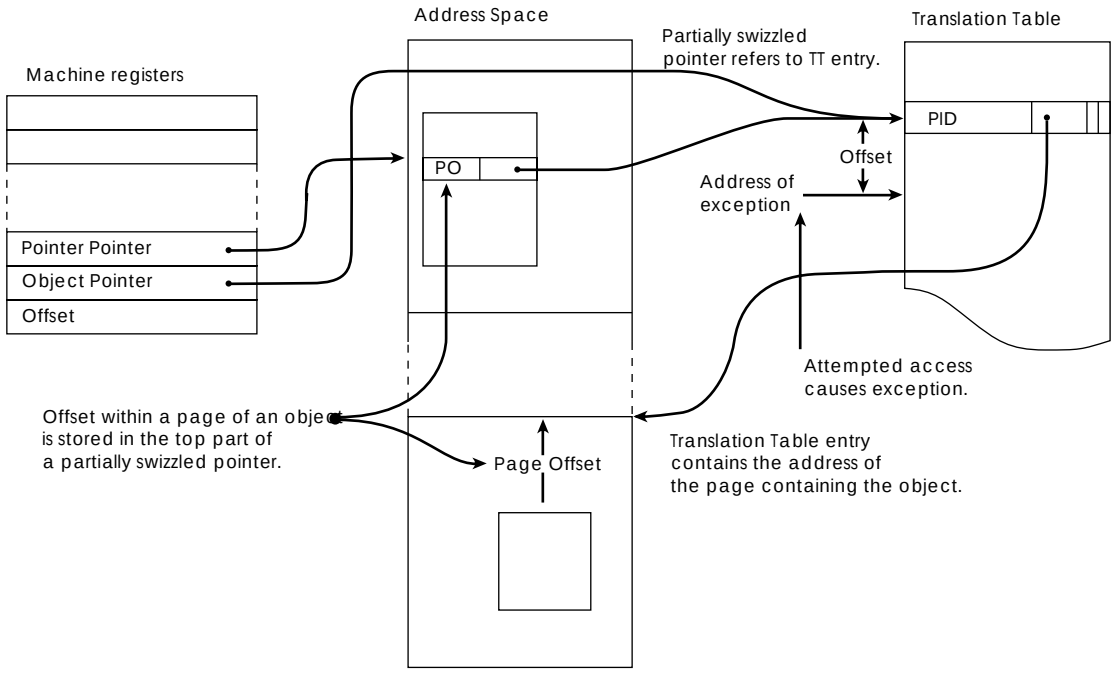

Figure 10: Pointer dereference via a partially swizzled pointer

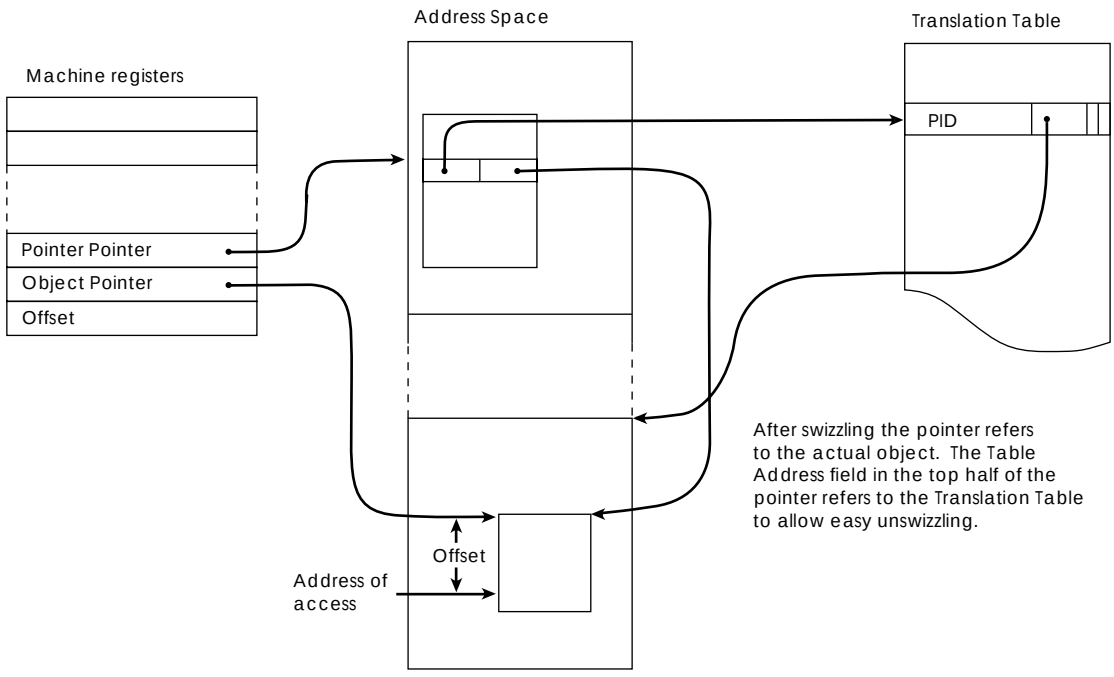

Figure 11: Dereference after the completion of swizzling

The scheme described above is directed at those processor architectures that only support simple addressing modes and require a number of instructions to carry out a dereference. Some processors are capable of executing the sequence described above in a single instruction, on such architectures the exception handler can decode the instruction pointed to by the saved PC. Such a scheme can be more flexible for two reasons. Firstly it may use many different addressing modes and secondly, it is not necessary to designate

particular registers since the instruction will indicate unambiguously which registers are being used and for what purpose. However, the complexity of the exception handler is higher.

4.3.2. Pointer comparisons

Since a pointer can exist within the system in one of two forms, care must be taken with pointer comparisons. Pointers can either be partially swizzled in which case they contain an offset and a reference to their translation table entry, or fully swizzled, in which case they contain a translation table reference and a pointer to the actual object. These two forms can be differentiated since the translation table and object area occupy distinct address ranges. In both formats a reference to a translation table entry, and the page offset is present, it is therefore enough to compare these values when performing pointer comparison.

4.3.3. Large Objects

Objects which cross page boundaries and more importantly very large objects which span a large number of pages require no special treatment. When an object spans more than one page it is not necessary for the whole object to be resident at one time. However it is necessary to reserve enough virtual memory to hold the object in a contiguous span so that it is possible to fault the rest of the object into memory as it is required. This preserves all the advantages that a demand paged virtual memory space has for sparse access to large objects.

4.3.4. Management of the translation table

The scheme described uses a translation table similar in format to that used by Wilson. Whereas translation tables in Wilson's scheme are of fixed size, only describing pages in the machine address range, our scheme requires a table that provides entries for every page that is referenced by pointers within virtual memory. Growth of the translation table takes the place of greedy allocation of virtual memory in Wilson's scheme.

The table has two major constraints placed upon its organisation: firstly, translation table entries are referenced directly by objects, therefore table entries may not move. Secondly, the action of swizzling pointers requires that it is possible to find entries from their PID quickly, otherwise the swizzling on page fault becomes a performance bottleneck.

Since pages are removed from the virtual address space, the translation table will eventually contain entries for pages which are no referenced from the virtual address space. By a simple modification to the scan used to deswizzle pointers during reclamation of virtual address ranges, these stale entries can be garbage collected. Any pointers found during the scan may be followed and the mark bit set in the referenced translation table entry. Once the scan has completed, the translation table is scanned and those entries without a mark bit set may be reclaimed. During this scan partially swizzled pointers for which the referend is resident may also be swizzled. Thus the reclamation pass through memory results in all references to resident objects being fully swizzled, stale entries in the translation table being eliminated and the freeing of virtual memory.

4.3.5. Creation of new objects

Many objects are created during the execution of user code; many of those objects will be short lived and therefore not require the allocation of a PID. Objects only require a PID when they become visible outside of the virtual address space in which they were created. In practice, this means an object that already has a PID acquires a reference to them.

We now describe a scheme whereby the allocation of PIDs is performed at the latest possible time. Pointers to new objects only contain the object's address; the field that would ordinarily refer to the translation table address is set to a sentinel value that indicates that the object does not yet have a PID allocated. When a page is deswizzled, pointers to objects without PIDs will be detected. At this time, a PID is allocated and a translation table entry created.

4.3.6. Exception Handlers

Clearly one of the main performance determinants of this scheme will be the performance of the exception handling mechanism. Conventional operating systems can provide a platform with which to prototype a system such as we have described. However they place a large overhead on the user program, typically over 10,000 machine cycles per exception. Where the designer of the system has control of the hardware and is able to define the actions of the exception handler the overhead can be as low as a dozen machine cycles. The architecture described in this paper is of most benefit in an operating system designed from scratch to support it.

4.3.7. Access to the translation table.

In the hybrid architecture described in this paper user code is prohibited access to the translation table whilst the exception handler and page fault handler have full access to it. This situation is also found in some garbage collection schemes [11] and the solutions are the same. If the exception and fault handlers are implemented within the kernel they can make use of the full access accorded the kernel to user address spaces. Alternatively it is possible to place the translation table within the user's virtual address space but to have a protected area of the same size at high memory to which all the partially swizzled pointers refer. When interpreting pointer values during swizzling and deswizzling the offset between the translation table and the protected area is subtracted from the pointers to provide the actual address within the translation table. This allows the system to be implemented without modifying the operating system kernel.

5. Comparison of the schemes

The following table summarises the main design features and costs of each of the three schemes described.

- *Granularity* is the size of the entity which the swizzling scheme manages.
- *Code compatibility* lists those areas in which specific changes to the code running on the system must be made.
- *Dereference overhead* is the extra cost (if any) of performing a dereference operation.
- *Assignment* is the size of the data assigned in pointer assignment.
- *Object fault overhead* lists the main activities that must be performed when a reference to a non-resident object occurs.

- *Recovery of VM* lists what actions are required when virtual memory is exhausted.
- *Recovery of Translation Table* lists what actions are required when space for the Translation Table is exhausted.
- *VM space allocation* lists the entities for which virtual memory must be allocated.
- *VM space used* lists the entities for which virtual memory is used to hold data.
- *Translation Table allocation* lists the objects for which an entry in the *Translation Table* must be made.
- *Deswizzle action* compares the costs of deswizzling a pointer.
- *Stabilisation Action* lists the actions required to stabilise the state of the system to persistent storage.
- *Large object overhead* compares the use of virtual memory to hold large objects.
- *Sensitivity to exception handler speed* compares how performance is affected by the exception handling mechanism.
- *Overall VM space* compares the use of virtual memory of the systems.

Feature/System	CPOMS	Wilson	Hybrid
Granularity	Object	Page	Page
Code compatibility	Software check per dereference	No implications	Use of defined sequence for dereference, and pointer comparison
Dereference overhead	Software check per dereference, possible swizzle	None	Usually none, possible swizzle
Assignment	Virtual address	Virtual address	Twice virtual address
Object fault overhead	Copy single object from store	Copy page from store and swizzle internal pointers. For each new referenced page, interrogate store and allocate memory	Copy page from store, swizzle internal pointers and follow pointer chains if used
Recovery of VM space	Rebuild system if VM exhausted	Invalidate VM and rebuild	Invalidate VM and rebuild
Recovery of translation table	Rebuild system if PIDLAM exhausted	Fixed size table	Garbage collect translation table
VM space allocation	Accessed objects	All referenced pages	Accessed pages
VM space used	Accessed objects	Accessed pages	Accessed pages
Translation table allocation	Accessed objects	Entry per page of VM	Entry per page of VM
Deswizzle action	Follow pointers to PID stored with object	Search translation table for object entry	Follow pointer to translation table
Stabilisation action	Per modified object: Deswizzle pointers, write object to store	Per modified page: Deswizzle pointers, write page to store	Per modified page: Deswizzle pointers, write page to store
Large object overhead	Entire object kept in virtual memory	Accessed pages kept in virtual memory	Accessed pages kept in virtual memory
Sensitivity to exception handler speed	Little impact	Slight impact	High, less when swizzle chain is used
Overall VM space	Lowest	Highest	Low

Each of the three systems described has particular strengths. The CPOMS design is the most parsimonious in the use of virtual memory, but also the one with the highest run time overhead. Wilson's design has the lowest running costs when not page faulting, but the highest page fault costs. If the amount of virtual memory used becomes large Wilson's scheme must incur the cost of rebuilding the working set and expense of an extra translation table. Hence Wilson's design is probably best suited to environments small enough for it never to be necessary to recover allocated virtual memory. Applications with shorter lifetimes and smaller data bases would be most suitable. The hybrid scheme has running costs similar to that of Wilson's design, has lower page fault costs, and is able to recover virtual memory and translation table space more easily. This is at the cost of forcing the use of a special dereference instruction sequence, and double length pointer assignments.

Pointer swizzling may be characterised by the time at which: pointers to be swizzled are encountered, translation table entries are allocated, memory for the object is allocated, an object is loaded from the store, the initial pointer that refers to the object is swizzled. Further characterisations are: whether other instances of the pointer to the same object are swizzled at the same time, and whether pointers within objects newly faulted into memory are swizzled to refer to resident objects. Each of these activities may be performed either eagerly or lazily, the following table summarises the characteristics of the three systems described.

Feature/System	CPOMS	Wilson	Hybrid
Locate pointers	Lazy	Eager	Eager
Translation Table allocation	Lazy	Eager	Eager
Allocation of VM	Lazy	Eager	Lazy
Object Loading	Lazy	Lazy	Lazy
Swizzle to VM Address	Lazy	Eager	Lazy
Swizzle other pointer instances	Lazy	Eager	Eager/Lazy
Swizzle new pointers	Lazy	Eager	Eager

6. Conclusions

This paper describes three architectures capable of supporting arbitrarily large persistent identifiers and large object stores using conventional hardware. Two of these represent opposite ends of a design spectrum; the third is a new hybrid architecture which embodies useful attributes of the other schemes and which has some useful attributes in its own right. The hybrid architecture maintains the advantages of lazy swizzling found in the CPOMS design namely only allocating space for objects, and fetching objects, when they are referenced. The hybrid design also maintains the advantages of page based designs, requiring no runtime checking of pointers and allowing sparse references to large objects without the need to copy entire objects into virtual memory. A design for machine level dereferencing has been presented that allows exception handling code to swizzle pointers

on demand without requiring checking by user code. Many of the techniques described in this paper may be of benefit in other designs.

Acknowledgments

This paper benefits from discussions with Malcolm Atkinson, Ron Morrison, John Rosenberg, Sándor (Alex) Farkas and Kevin Maciunas. For those discussions we thank them. We would also like to thank Tracy Lo Basso, Bett Koch and Andrew (Noid) Cagney for their comments on an earlier draft of this paper. This paper was completed despite the arrival of Graham Stewart Dearle on 3/7/92. This work is supported by ARC Grant number A49130439.

References

1. "PS-algol Abstract Machine Manual", Universities of Glasgow and St Andrews, PPRR-11-85, 1985.

2. "PS-algol Reference Manual - fourth edition", University of Glasgow and St Andrews, Persistent Programming Research Report 12/88, 1988.

3. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, vol 26, 4, pp. 360 - 365, 1983.

4. Atkinson, M. P., Bailey, P. J., Cockshott, W. P., Chisholm, K. J. and Morrison, R. "POMS: A Persistent Object Management System", *Software Practice and Experience*, vol 14, 1, pp. 49-71, 1984.

5. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, vol 17, 7, pp. 24-31, 1981.

6. Brown, A. L. "Persistent Object Stores", Ph.D Thesis, Universities of St. Andrews and Glasgow, 1988.

7. Brown, A. L. and Cockshott, W. P. "The CPOMS Persistent Object Management System", Universities of Glasgow and St Andrews, PPRR-13, 1985.

8. Cockshott, W. P., Atkinson, M. P., Chisholm, K. J., Bailey, P. J. and Morrison, R. "POMS: A Persistent Object Management System", *Software Practice and Experience*, vol 14, 1, 1984.

9. Cockshott, W. P. and Foulk, P. W. "Implementing 128 Bit Persistent Addresses on 80x86 Processors", *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information*, Bremen, West Germany, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 123-136, 1990.

10. Dearle, A., Rosenberg, J., Henskens, F. A., Vaughan, F. A. and Maciunas, K. J. "An Examination of Operating System Support for Persistent Object Systems", *25th Hawaii International Conference on System Sciences*, vol 1, IEEE Computer Society Press, Poipu Beach, Kauaii, pp. 779-789, 1992.

11. Ellis, J., Li, K. and Appel, A. "Real-time Concurrent Collection on Stock Multiprocessors", DEC SRC, 25, 1988.

12. Kaehler, T. and Krasner, G. "LOOM – large object-oriented memory for Smalltalk-80", *Smalltalk-80: Bits of History, Words of Advice*, ed G. Krasner, Addison-Wesley, pp. 251-270, 1983.

13. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. "Cache Coherence and Storage Management in a Persistent Object System", *Proceedings, The Fourth International Workshop on Persistent Object Systems*, Marthas Vineyard, ed A. Dearle, G. Shaw and S. Zdonik, Morgan Kaufmann, pp. 99-109, 1990.

14. Kolodner, E., Liskov, B. and Weihl, W. "Atomic Garbage Collection: Managing a Stable Heap", *Proceedingss of the 1989 ACM SIGMOD International Conference on the Management of Data*, pp. 15-25, 1989.

15. Lamb, C., Landis, G., Orenstein, J. and Weinreb, D. "The Objectstore Database System", *CACM*, vol 34, 10, pp. 50-63, 1991.

16. Moss, J. E. B. "Working with Persistent Objects: To Swizzle or Not to Swizzle", COINS, University of Massachusetts, 90-38, 1990.

17. Rosenberg, J. "Architectural Support for Persistent Object Systems", *International Workshop on Object-Orientation in Operating Systems*, IEEE Computer Society Press, Xerox-Parc, California, 1991.

18. Wilson, P. "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", *ACM Computer Architecture News*, June, pp. 6-13, 1991.