A Survey of Architectures for Memory Resident Databases

Gordon Russell Paul Cockshott

May 13, 1993

Department of Computer Science
University Of Strathclyde
26 Richmond Street
Glasgow
G1 1XH
Scotland

Abstract

Persistent object oriented architectures have been researched for many years, deriving initially from the Manchester University Atlas machine. In reality however, few actual implementations of persistent architectures exist. In the first half of this paper an examination of four well known designs is examined, namely the SYSTEM/38, MONADS, MUTABOR, and the Rekursiv. Each machine's object management model is explained, along with an analysis of the design decisions made. Following this, a discussion concerning the ideal persistent architecture is presented, suggesting design decisions which should be considered in any future persistent architecture.

1 Historical background

The idea of architectural support for persistent programming derives ultimately from the work of Tom Kilburn and others [1] on the Manchester University Atlas computer. They introduced the idea of what was termed a single level store: a notion that is now more familiar to us as virtual memory.

During the 1950s machines used a variety of different store technologies: Williams Tubes, mercury delay lines, magnetic cores and moving magnetic devices. Although these media differed considerably in their response times, they were all used in the same conceptual fashion, as the primary store of the machine.

For instance, an earlier machine by Kilburn at Manchester [2] had combined, the then very new, transistors with a magnetic drum for its main instruction store. As a consequence of using drums its instruction cycle was slow at some 30 milliseconds, but this was partially offset by the cheapness of drum store in comparison to its competitors. The single level store of Atlas was introduced as a means of making a drum store perform at almost the same speed as the more expensive cores. The drum continued to act as the main store, but pages of it were automatically transferred on use to one of a small number of page frames implemented as magnetic cores.

At this early period of computer development, computers were still seen primarily as number processing machines rather than as repositories for long term data. Although, by 1960, all the main memory technologies were based on magnetic effects and thus non-volatile, this aspect was not seen as being of any great significance. The drums that provided the main store of the Atlas were not used to hold long term information. Data to be processed was kept off-line on tapes.

The emphasis changed with the development of disk stores, which were from the start seen as long term stores. In consequence, when people first tried to develop interactive, virtual memory operating systems such as Multics [3] and Emas [4], they attempted to integrate disks and short term memory into a single level store. Ideally one would have liked to incorporate all of the disks as part of the permanent address space of the machine. In that case a disk file would just be a particular range of addresses.

Two problems prevented a simple implementation of this approach:

- The size of the disks exceeded the address space provided by the underlying machine architecture.
- If a fixed range of addresses were allocated to a file, it was impossible to allow the file to grow.

The EMAS system, for example was initially implemented on ICL machines whose basic architecture was a copy of the IBM 360 series [5]. This restricted its address space to 24 bits or 16 megabytes, far too little for a disk farm. The answer adopted was to allow files to be temporarily mapped into the address space of a process, a method that has recently been included in implementations of Unix [6].

The answer to the problem of mapping dynamically growing files, could in principle be solved by segmentation as was advocated by Iliffe [7], and used in Multics and the later versions of EMAS that were implemented on the 2900 series [8] machines. A file could now be made equivalent to a segment and allowed to grow up to the maximum size of a segment, but this just created new problems. The segments were smaller than large files, and there were fewer segments available than there were files.

This early experience with operating systems showed that graceful integration of non-volatile store into machine architectures required:

- A segmented architecture
- A large number of segments
- A large segment size

These have been the goals of the designers of more recent persistent store computing machines. We shall investigate how far they have succeeded in our description of four persistent systems: IBM SYSTEM/38, MONADS, MUTABOR and Rekursiv.

1.1 SYSTEM/38, AS/400

The IBM SYSTEM/38 [9] [10], known in its more recent models as the AS/400 was introduced at the end of the 1970s. At this point, IBM were becoming aware of the limitations in the virtual memory architecture supported by their existing mainframes. Their principle fault was to provide too small an address space; an architectural mistake more difficult than any other to remedy. The IBM engineers took the bold step of more than doubling the address width, from 24 bits on the 360 series to 64 bits on the SYSTEM/38.

Following the developers of Atlas, the SYSTEM/38 team called their memory model single-level store, but with the advance in storage capacity that had occurred in the in the intervening decade and a half, the emphasis was different. The purpose of the single level store was now to integrate a potentially vast disk database into the virtual address space of the computer. The difference between this and the conventional view of virtual memory is clearly brought out in figure 1.

All store, not just main memory is controlled by the virtual memory manager. No input/output to auxiliary disk store is ever performed by user programs. Indeed, there is no concept of a disk file to which I/O could be directed. Instead, the store is grouped into logical objects which persist. Objects persist until explicitly deleted by the user or until the next IPL in the case of certain operating system specific objects. Object management and virtual memory are supported in microcode.

1.1.1 Object Addressing

Objects are addressed using 16 byte identifiers termed *pointers*. A pointer consists of an 8 byte object address plus additional information about the status of the object, authorization properties and the object's type.

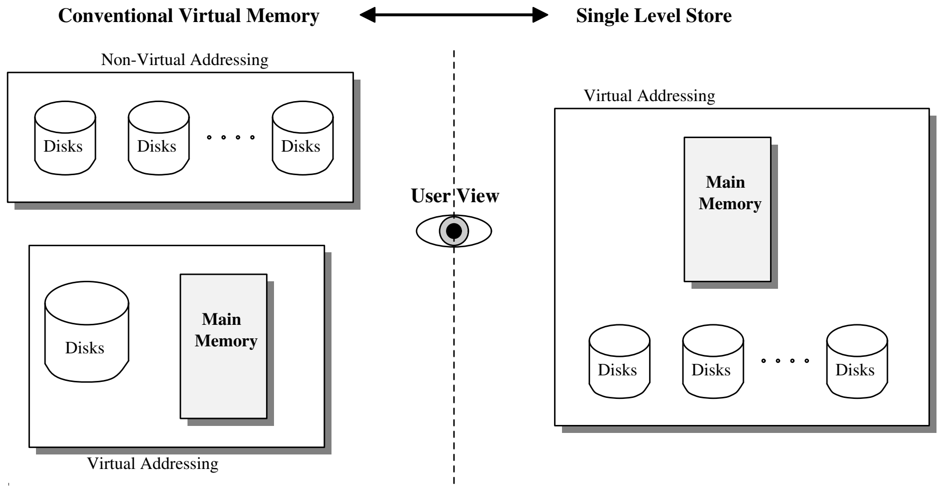

Figure 1: Two Views of Virtual Memory

The store of the machine is tagged, with tag bits being used to determine if a 16 byte aligned sequence is a pointer. If it is, the hardware imposes restrictions on what can be done with it to prevent the corruption or overwriting of pointer information. The layout of a pointer is shown in figure 2.

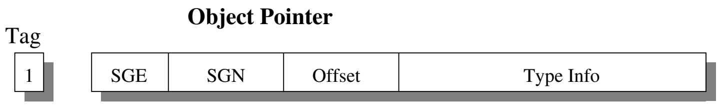

Figure 2: A SYSTEM/38 Pointer

The address part of the pointer consists of 3 fields:

- *Offset*: 24-bit byte offset within a segment group
- *SGN*: 24-bit segment group number
- *SGE*: 16-bit segment group extender.

Objects are each assigned a segment group, allowing 16 million objects to be active at any one time, each of which can be up to 16 megabytes in size.

The segment group extender is an error checking mechanism associated with the re-use of object numbers. A maximum of 2^{24} objects is not enough to avoid the need for segment groups to be reused. When an object is explicitly deleted (the machine does not use garbage collection), the segment group associated with it may be reused. When, however, this takes place, the new object address will be given a different segment group extender. The extender is also written into the header of the segment group. On the dereference of a pointer, a comparison is made between the extender field of the pointer and that in the segment group header. Any difference, indicating that the pointer referred to an earlier object that had occupied the same portion of the virtual address space, triggers a fault.

The hardware currently uses 48 bits of the virtual address to obtain a physical address. For this purpose, the 48-bit address is considered as a 39-bit page address and a 9-bit offset into a 512 byte page. The small page size is interesting; it indicates that the space overheads, associated with transferring a small object into memory, would be less serious than on a system with larger pages.

1.1.2 Address Translation

When the number of pages in the address space is very large, 2^{39} in the case of the AS/400, conventional hierarchical page tables become impractical. Consider what would happen if one tried to map the address space hierarchically. . .

A single 512-byte page would hold 64 or 2^6 page-descriptors of 64 bits each. We would thus need some 2^{33} pages to fully map the address space. Of course one need not map it fully, one only needs mapping pages for pages currently resident, but an object address space is likely to sparsely used. Each object occupies a segment group taking up 16 megabytes of address space. This alone requires 3 levels of mapping tables, so the smallest resident object would tie down 3 page frames in mapping information.. In addition to the space constraints, there are speed considerations. some 6 or 7 levels of indirection would be involved in converting a logical to a physical address.

To avoid these problems the notion of the inverted page table was developed independently for the SYSTEM/38 and for MONADS [11] [12]. The technology was later adopted on the IBM 801 experimental RISC processor [13] and the RS/6000 series [14]. An inverted page table contains records for each physical page of RAM indexed on the virtual pages that they contain. An indication of how it works is provided in figure 3.

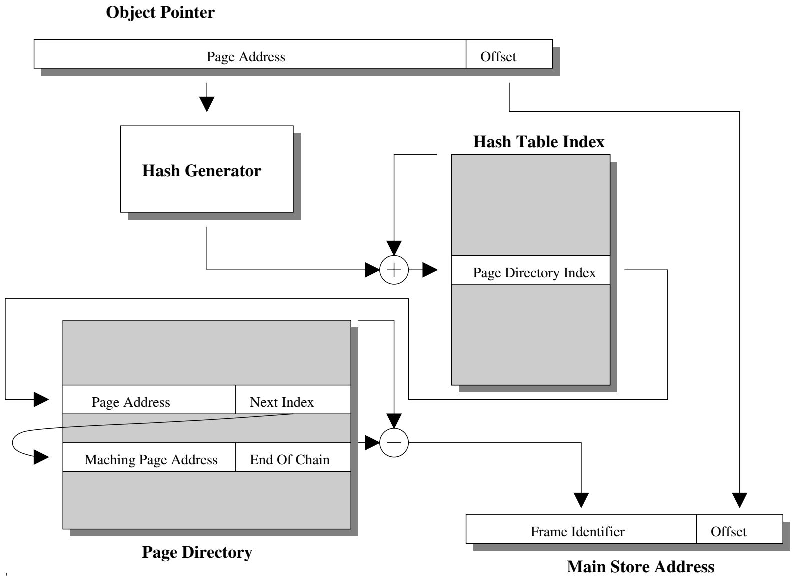

Figure 3: The Inverted Page Table

1.2 MONADS

The MONADS project was first established in 1976 to investigate techniques to improve on the design and development phase of large software systems. From that point it has developed into an object oriented environment, utilizing a segmented virtual paged memory hierarchy. This environment has taken a number of different forms over the years, including Hewlett Packard 2100A minicomputers (MONADS-I and MONADS-II), a custom micro-programmable processor (MONADS-PC), and a SPARC based system (MONADS-MM), of which there are two current implementations; the MONADS-PC (Personal Computer) [15] and the MONADS-MM (Massive Memory) [16]. These two machines can be programmed in a number of high level languages, including a PASCAL dialect and, to a limited extent, LEIBNIZ [17].

There are three major design philosophies which underline the MONADS environment:

1. Module Structure. This was desirable since many of the programming languages considered for use on the MONADS system were scope based. The exportation of local variables from modules is therefore strictly forbidden, forcing access to such information through procedural-based mechanisms.

2. Process Structure. The whole approach to operating system composition follows the 'in-process' model [18] for service requests[1]. Although Keedy [19] argues that such a mechanism has many inherent advantages in a dynamic system, the domain switching and domain protection required to handle this type of design satisfactory generally requires significant hardware support.

3. Uniform Virtual Memory. It is desirable, from a programmers viewpoint, to provide either exactly one way to perform a number of logically similar tasks, or to provide a number of ways where each mechanism can be done equally well for each of the tasks. In a non-uniform system, the processes involved in locking an object and locking a file are not identical, even though they are logically similar. Uniform virtual memory does not exhibit the divisions found within non-uniform solutions. Such uniformity lies at the heart of the quest towards persistent object oriented systems.

In evaluating the MONADS design strategy, careful consideration of the object management system should be performed, including the effects of its block-type information hiding philosophy.

1.2.1 MONADS Memory Management

The fundamental user-level[2] memory addressing mechanism used in MONADS is the virtual memory address. This address is similar in design for both the machine variants, with the PC utilizing 60 bits and the MM 128. The layout of the virtual address can be seen in figure 4.

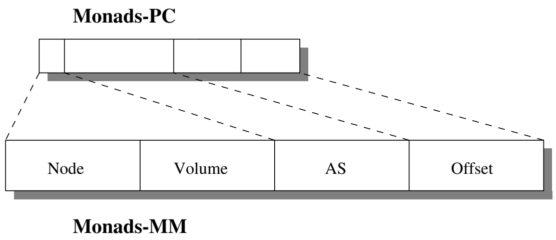

Figure 4: Virtual Address for MONADS

From figure 4, it can be seen that the virtual address is split into four component parts, *i.e.* the node, volume, AS and offset.

- The virtual memory node number indicates the machine's network address on which the desired physical memory slot is located. This is 32 bits long for the MONADS-MM machine. This may not be a sufficiently large enough range of values, especially considering that some network addressing systems now make use of 48 bit address fields (remember that many network routers rely on a hierarchical physical location description being stored within the network address, thus the resulting address description is sparse).

- Disks within MONADS can be identified by one or more volume numbers. If more than one volume number refers to a single disk, then each number represents a partition within that disk. This field is 32 bits long for the MONADS-MM machine. However, this field with the node

[1] Procedure-oriented or 'in-process' techniques invoke operating system **instances** via procedure based access, as opposed to 'out-of-process', which communicates with operating system **tasks** through communication channels.

[2] Although clearly the fundamental memory addressing mechanism for the underlying machine is the physical address.

number is represented by only 6 bits in the MONADS-PC system. This is clearly insufficient for any realistic networked system.

- Persistent memory is split up into a number of separate ‘chunks’, with each of these chunks given a unique identifier. These chunks are known as *address spaces*, identified by address space numbers. Each of these spaces are divided up into a number of fixed sized pages. Note that these address space numbers are never reused, even after the memory chunk has been deleted.

- In order to access data elements within each address space, an offset can be provided such that address plus offset identifies a unique memory location.

With the virtual address partitioned in such a rigid manner, the virtual addresses become fragmented (especially since virtual addresses are not reused under MONADS). However, since the range of virtual addresses are deliberately made much greater than the range of physical memory addresses (and that even at the maximum rate of usage the virtual addresses will not be used up during the lifetime of the system), this does not pose a serious problem.

Note that the virtual addresses are not directly used by executing tasks. Instead, objects (or segments as they are known in MONADS) are accessed through object identifiers. These identifiers refer to object capabilities (similar to IBM SYSTEM/38 capabilities) [20], and form the basis for object protection under MONADS.

1.2.2 MONADS Segment Capabilities

All segments used in MONADS are referenced via capabilities. Each segment is split into a number of parts (as in figure 5). The first part of a segment is known as the *segment control section*. This contains the actual size of the segment and details describing its contents. The actual data part of the segment is called the *information section*. It is here that information can be stored and retrieved, based on the segment control data. The final part of a segment is referred to as the *segment capability section*. Within this area references to other segments can be placed so as to permit arbitrarily complex graph structures to be represented on the system.

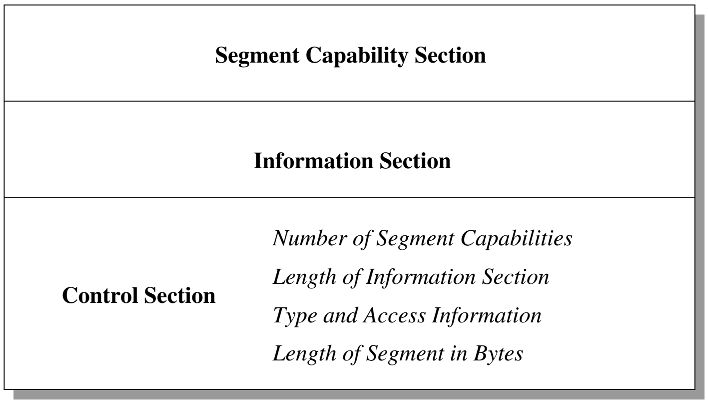

Figure 5: Segment Layout in MONADS

This type of fixed segment layout contains limitations which increases the implementation complexity of object oriented languages. This problem can most clearly be observed with respect to type inheritance. In typical object oriented languages, inheritance is supported by combining a current segment type with a new segment (which contains the supertype information). The combining process is often achieved by tagging the new segment at the end of the older one. This is not possible in MONADS, where it is impossible to have an arbitrary combination of data and capabilities. Inheritance may be implemented by using some kind of indirection from a 'higher level' segment, but this clearly involves the expense of performing the initial indirection.

To refer to information contained within any segment, all that is required is the address of the segment (all other information concerning the segment being self contained). For efficiency, registers, known as *capability registers*, are used when referring to segments. Reference to an element within a segment via such a register is depicted in figure 6.

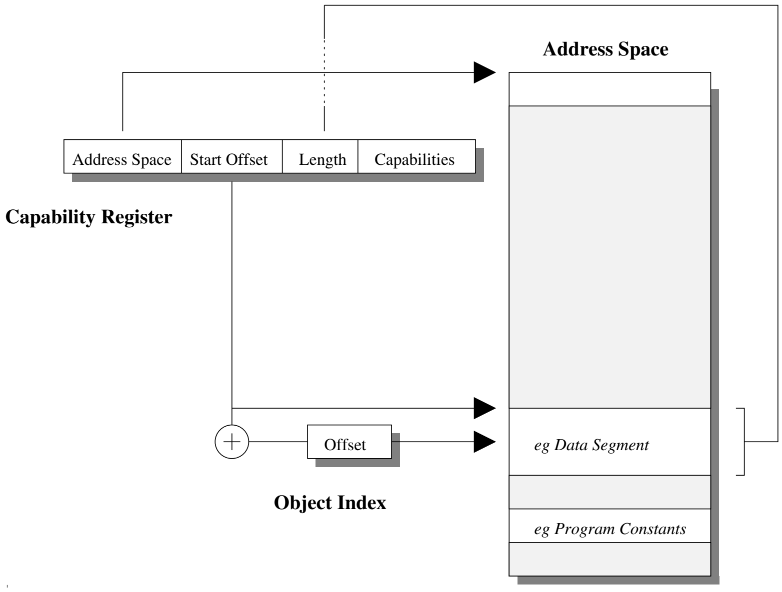

Figure 6: Object Mapping in MONADS

By accessing a segment, and loading a capability from that segment, all accessibly data structures can be traversed. Loading of the capabilities must clearly be protected from malicious users, thus a special machine instruction is provided for this function. Indeed, another instruction performs the modification of the segment capabilities. The root of all addressing is found within a special segment (one per process), called the *base table*. An additional machine instruction is included to allow access to the contents of this table. Note that such instructions obviously interfere with instruction set orthogonality and thus add to compiler complexity.

Segment capabilities are constructed such that all segment addresses are expected to reside within the current address space [21]. This has the effect of saving memory space (and saves on the effort required during garbage collection), at the expense of limiting the scope of data structures. It is therefore advisable to group related objects (*e.g.* segments) together within a single address space. Segments outside of the current address space can however be referenced by using indirect capabilities, which themselves point to full virtual addresses.

1.2.3 Module Management

There are two distinct types of objects recognized by MONADS; segments and modules. Modules present a procedural interface to other external modules. Using modules, it is possible to implement such constructs as traditional 'files', where the module manager can present an interface which mimics traditional file access. In fact, since MONADS describes itself as persistent, no files actually exist. However, MONADS deviates from more traditional persistence in that capabilities referring to a segment may not be passed to other modules and freely retained.

The reason for non-persisting capabilities external to the owner module lies with a fundamental limitation with the MONADS environment. In the attempt to remove the need for a central mapping table [21], compounded by the desire to allow segment slices to be passed between modules (in a controlled way), the problem of virtual memory limitations becomes a factor.

With the removal of the central mapping table, virtual memory can not be easily reused. This is especially true if capability revocation is attempted (as is possible with when capabilities are copied by modules external to the segment owner). To counteract this problem, capabilities are only permitted to be stored on the calling stack of external modules (thus are automatically deleted on a return from that module). Thus revocation of segment capabilities does not occur, and virtual memory can be safely reused.

Due to the grouping of segments within modules, the individual segments are no longer considered to be separate objects with a type associated to each, but instead as a collection of similar objects which are accessed via the procedural interface of the module. This style of data modelling is typical of architectures which use a set of capability registers to hold the roots of a relatively small number of large entities.

1.2.4 MONADS Paging

Each of the allocated memory address spaces must be wholly stored within a single volume (although a volume will typically carry more than one address space). For efficient paging, a disk page table is needed for each address space, and this is held in a protected region of the address space which it describes. A single root page exists for each volume, and points to the volume directory, which in turn refers to each of the address space paging tables.

Every MONADS node contains an Address Translation Unit (ATU), which maps the virtual addresses onto physical memory locations. The ATU is implemented as an inverted page table (as in the case of the AS/400), constructed from hardware, such that its size is proportional to the size of the physical memory map. If the page required does not exist in physical memory, a page fault is then generated. At this point two separate routes can be followed: either the faulted page is within the current node (in which case the page can be loaded off disk), or the page is stored within some other node (resulting the local node asking the networked node for the page).

On a locally resolvable address fault, the disk address of the desired page can be read from its primary page table. This table is indexed with bits 12-27 of the virtual address offset value, and thus contains 2^{16} entries of 16 bit disk addresses. It may be that this primary page table does not exist either, and therefore also generates an address fault. This results in the secondary page table for that address space being accessed, which contains 32 disk address entries (in the MONADS-PC system). Thus each entry corresponds to a single page of the primary page table ($(2^{16} \times 2)/4\text{K bytes} = 32$). The required section of the primary page table can now be loaded from disk. It is also possible that the secondary page table for the address space is not present either. The disk address of this page can be found by accessing the root page table, which is always found at address space zero of each volume. Part of this address space paging structure can be seen in figure 7.

The layout (at least with respect to the MONADS-PC version) of this disk address tree is such that, for address spaces smaller than 256 memory pages (with each page containing 4K bytes), the entire tree can be found within the first page of the address space. Indeed, for address spaces smaller than about 3.6K bytes the entire address space can be found within a single page (thus resolving the page fault in a single disk access) [22].

1.3 MUTABOR

From research conducted into the Profemo project at Gesellschaft für Mathematik und Datenverarbeitung mbH[3] (GMD), examining the design of reliable distributed architectures, came the design of MUTABOR (Mapping Unit for The Access By Object References). This forms an object-oriented

[3] The German National Research Centre for Data Processing

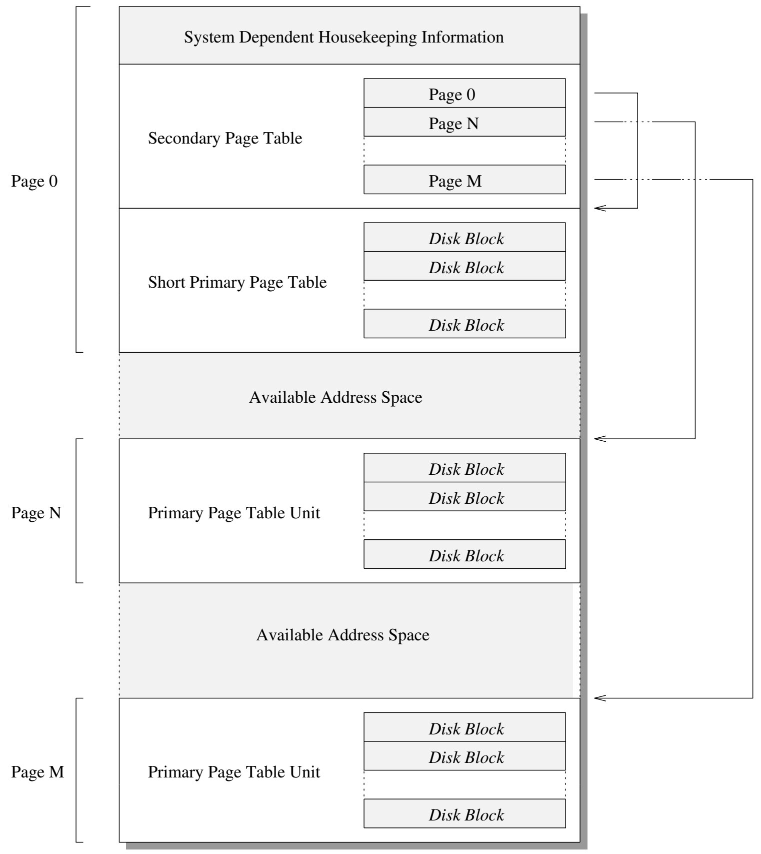

Figure 7: Layout of a MONADS Address Segment

architecture providing supporting for a secure and reliable computing base, via a layered, generalized transaction concept operating system. Dedicated hardware support is used for both object and transaction management [23].

MUTABOR consists of a microprogrammable coprocessor (which currently connects to the MC68020 coprocessor interface [24][pages 343,381-386], thus extends the MC68020 instruction set to include MUTABOR related functions) and an address translation cache (ATC). This ATC holds physical addresses of recently accessed objects, along with other information which is used in additional logic to authenticate access rights and size constraints.

In contrast to MONADS, MUTABOR implements a more fine-grained approach to object instances. Take for example a database holding student information (*e.g.* student name, address, exam marks, *etc.*) for two hundred students. Within MONADS, the entire two hundred entries would be stored inside a single module, supporting module interfaces such as *get student record* or *assign exam result to student*. In comparison, MUTABOR would implement each student record as a separate object, and each object would have associated type information evaluated at invocation time. It is assumed that the mean object size in MUTABOR is about 300 bytes [25], which compares favourably with other fine-grained approaches (such as 300 for iMAX [26] and 326 for Hydra [27]).

1.3.1 Object Store

The MUTABOR system implements object persistence within a long term repository known as the *passive space*. There also exists a second area, called the *active space*, which constitutes a virtual address space (mapped by the ATC) within which computations are performed[4]. If a persistent object which does not exist within the active space is referenced, then it is transferred from the passive space by some internal manager. Thus the active store simply functions as a virtual cache for the passive store. Since the transfer mechanism is automatic and transparent, the single level object store paradigm is maintained.

Additionally, within a fine-grained system, many transient objects (such as temporary local variables) are created and then quickly abandoned. Experiments from StarOS and Hydra show that this class of object forms about 95% of all objects instantiations [28][page 107]. It would be a severe system bottleneck if each of these objects required a PID (*e.g.* UID or very long Unique and persistent IDentifiers as they are known within MUTABOR). To reduce this overhead, each PID located within the active space is identified by an *object short name* (OSN). These OSNs are 24 bits long (in comparison to the 48 bit PIDs), thus allowing sixteen million objects to reside simultaneously within the active space. Note that since the OSNs are smaller than 32 bits, they can be fetched and stored within a single memory cycle.

The MUTABOR system does not directly interpret the object UIDs. If an object is not present within the active space, then the *object filter* manager is informed. It is the manager's duty to perform passive UID to active OSN transfer. A second manager, the *global object manager*, is used to resolve object requests which require object transfers between MUTABOR based nodes.

1.3.2 Memory Layout

In MUTABOR, every process is given an unique 16 bit process id number. This number is used to select a context specification for each system context (thus providing every process with a different view of the object store). Each context specification consists of four entries:

- The current context. This contains the capabilities for the current context, including temporary variables.
- Current root object, which points to the user defined object on which the current context (called a TSO or *type specific operation*) is operating on.
- The current object type, which allows access to all other TSOs for the current root object.
- The parameter object which gives access to the invocation parameter. This allows data and capabilities to be passed, thus simulating either call by value or by name.

A code invocation consists of the following form:

INVOKE (target root object,TSO-index,parameter)

with the target root object specified by a capability index (*i.e.* the process local name for an object) within the current context, the current root object, the current type object, or the current parameter object. This requires that a new context specification be created for the process (including the context's own capability list), and the seven-stage mechanism involved in creating the context can be found within [25].

In accessing an object via its capability index, entry zero of the context specification is indexed with the high order bits of the index, which selects a particular capability list. This list in turn is indexed by the remaining low order bits, which selects the capability for the object in question. Thus, each process has its own name for every capability, and this is known as the *local process object name* [29]. This mapping mechanism can be seen in figure 8 [25]. The capability produced in the figure is now used to access the required object.

[4] Note that *passive* and *active* are terms which are heavily overused, and as such tend not to be comparable across different systems.

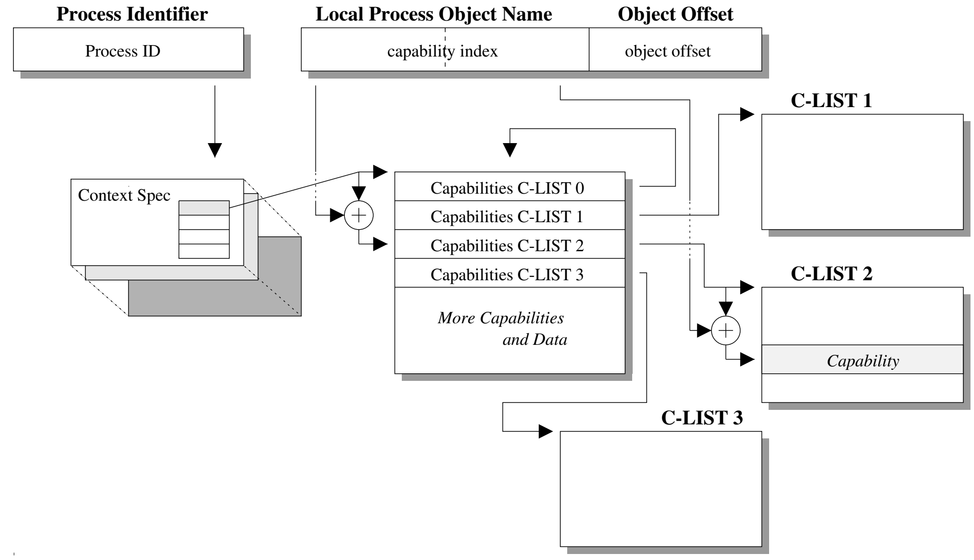

Figure 8: MUTABOR Context to Capability Path

The object OSN can be extracted from the selected capability: The high twenty bits of the OSN goes through a two level table to identify the page within which the object resides. Given the average object size of 300 bytes, the designers limited the mapping process to 16 objects per 4K page. Thus the last four bits of the OSN is used to select the required object header within each page. Additionally, this header contains information to check type and access rights dynamically at access time, thus allowing such facilities as object locking and shared access. This translation path is depicted in figure 9. Naturally, this method of object reference alleviates the need for a central mapping table (but creates a large data management task).

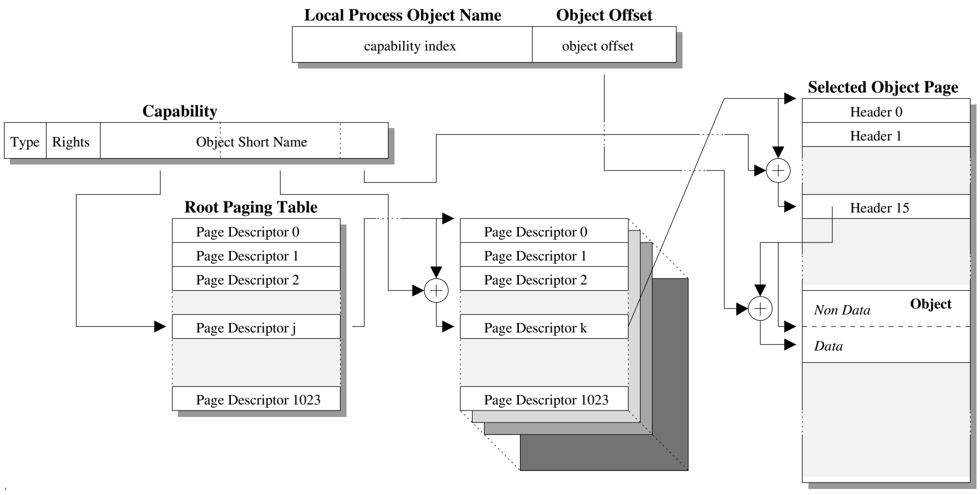

Figure 9: MUTABOR Capability to Address Path

Also from figure 9, the selected object is actually split into two parts: the data part and the non-data (or capability) part. Within the capability part lies the UID for that object, a link to the TSOs for that object, some system-required red tape, and any relevant capabilities required in representing complex data structures.

The object segregation is intended to protect objects from inadvertent or malicious accesses. Such segregation can be seen in a number of other architectures, such as MONADS, and is often known as *fenced-segmentation*. For MUTABOR, the capability part of an object can only be accessed by negative address offsets, which requires special coprocessor instructions to be used. Clearly, since MUTABOR and MONADS share a similar object layout scheme, the MUTABOR also exhibits the same limitations as MONADS with respect to type inheritance.

The need to maintain and traverse multilevel tables is an intrinsic overhead found in all similar types of object management implementations. It is for this reason that the ATC was introduced, containing the 4096 most recent address translations. The ATC is subdivided into 16 sections by 256 entries, with the 16 last most recently executed processes allocated one section each. This gives an estimated hit-rate of greater than 95%.

1.4 Rekursiv

The Rekursiv is a chip-set for the construction of object oriented processors designed by Linn, a Glasgow phonograph company, and produced by LSI Logic. A reasonably full description is given by Harland [30].

The complete chip-set allows the construction of micro-coded processors with object oriented virtual memory, the intention being for language developers to write special purpose micro-programs to support particular languages. The micro-instructions feature a high level of parallelism, controlling the simultaneous operation of three processing chips:

1. LOGIK the sequencer unit.
2. NUMERIK the arithmetic unit.
3. OBJECT the memory manager.

An overview of the system is shown in figure 10.

Associated with these processing units are seven distinct banks of special purpose RAM.

1. The CSMAP: a look-up table that maps opcodes to the addresses of the micro-routines that implement them.
2. The Control Store, which holds the horizontal micro-instructions.
3. The Control Stack, which holds linkage information both for micro-routines and high-level language routines.
4. The Evaluation Stack: a classic push-down store for the implementation of postfix instruction-sets and for the store of procedure-local variables.
5. The NAM or macro-instruction store.
6. The Page Tables, which map objects to physical addresses.
7. The Heap or Object Store into which currently active objects are loaded.

All but the last are implemented in high speed static memory. The motivation for providing all these distinct memories is speed. The ability to perform several memory accesses per clock cycle allows micro-code to be fast and compact. The complexity is somewhat less than it seems, since many microcoded architectures provide the equivalents of a CSMAP, Control Store, Control Stack and Page Tables (TLBs[5]), but hide them from view. Considering only what the assembly programmer sees, however, the Rekursiv still has an unusually complex store structure. Whereas on a von Neumann

[5] Translation Look-aside Buffer

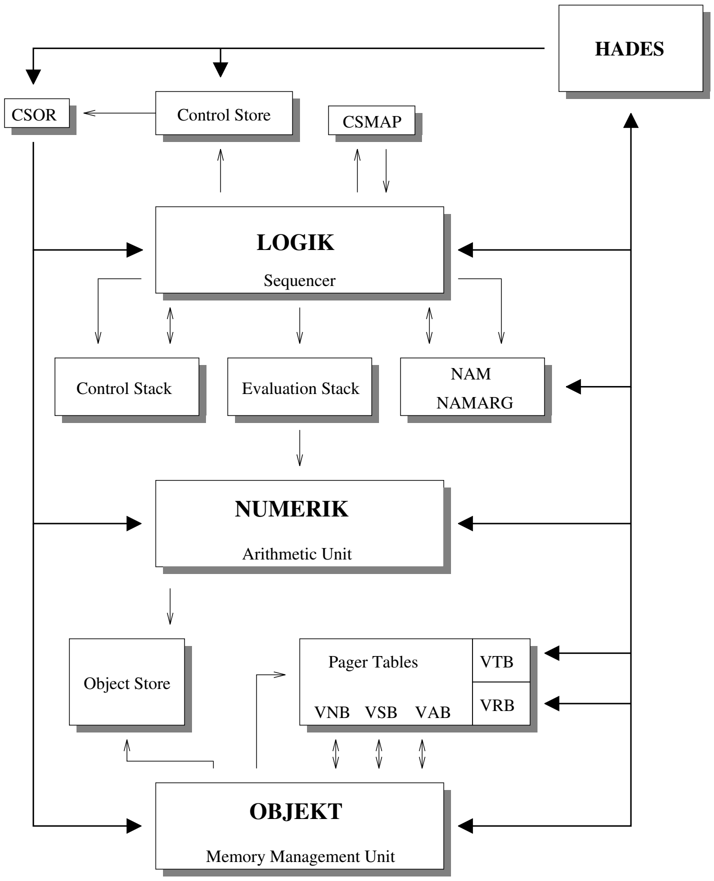

Figure 10: Overview of the Rekursiv

architecture, the evaluation stack, the heap and the macro-instructions would all share a common memory, the Rekursiv treats them as distinct. This has the advantage of allowing simultaneous pushing of a heap operand with instruction fetching, but it is questionable whether the complexities that this introduces for the operating system are worthwhile.

1.4.1 Memory Management

The Rekursiv is a 40-bit word machine, and its object identifiers are also 40 bits wide. Objects are sub-divided into two classes; compact and extended. Compact objects are those objects which are small enough to fit entirely within a single 40 bit word, and thus the entire object can be held within an object identifier. Extended objects tend to be larger than compact objects, and are stored at the particular memory to which the extended object identifier refers.

The format of these two object identifiers is shown in figure 11. Although their physical layout is distinct, the hardware contrives to give all objects the same abstract form. All objects appear as the sequence of fields shown in figure 12. They all appear as a vector of 40 bit words, the first three of which describe the object, with subsequent words holding the concrete representation. Figure 12 also shows five of the Rekursiv's CPU registers, and these correspond to:

- *VAR*: This corresponds to the *Value Address Register*, and contains the address of the first memory-bound element of the object.
- *VRR*: The *Value Representation Buffer* holds the word pointed to by *VAR* (*i.e.* the first element of

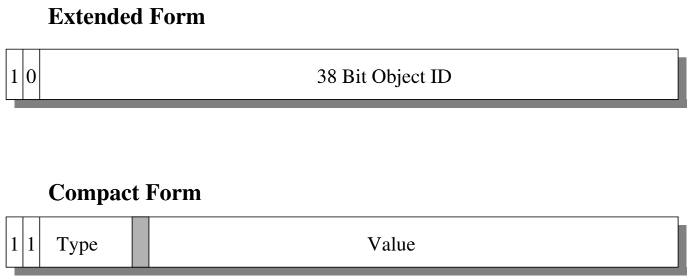

Figure 11: Two forms of Rekursiv Object Identifiers

the object).

- *VTR*: The type identifier of an object is held within this register (the *Value Type Register*).
- *VSR*: The *Value Size Register* holds the size in words of the object pointed to by *VAR*.
- *VNR*: This register (*the Value Number Register*) holds all 40 bits of the object identifier.

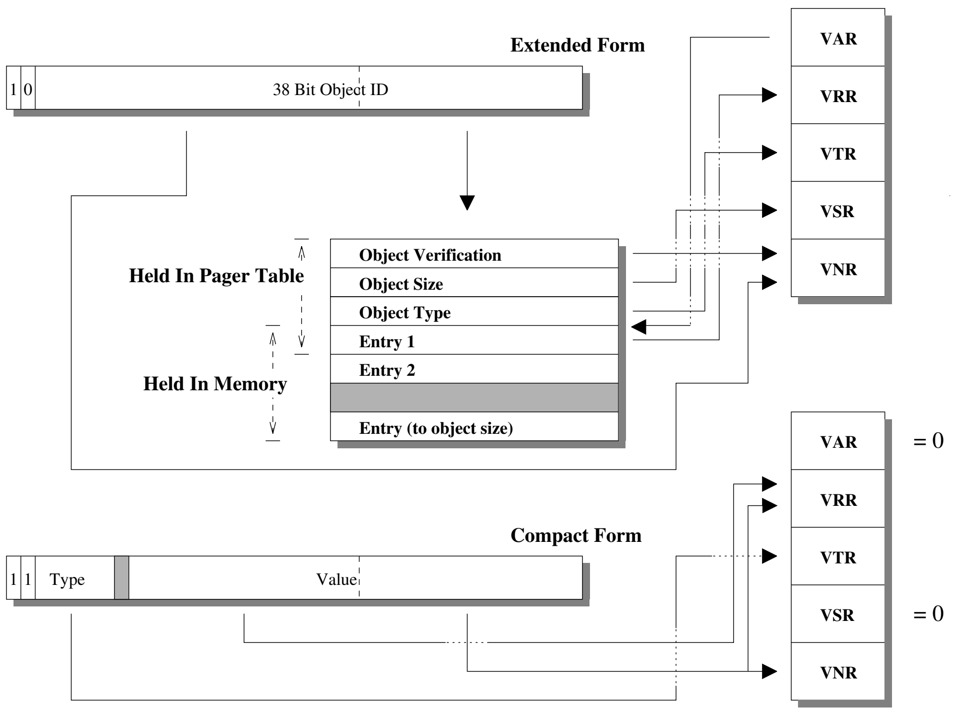

Figure 12: Rekursiv Object Register Fields

When an object is dereferenced, hardware in the memory management system loads this set of registers with the first four fields of the object {*VNR, VRR, VTR, VSR*} and its address {*VAR*}. In the case of compact objects both its size and address is always zero. The object's type is extracted from bits 33 to 37 of the object identifier, and *VRR* is taken from bits 0 to 31. In the case of extended objects, much of the required information is extracted from the page tables (figure 13) by the following algorithm:

1. The lower 16 bits of the object id are used to index the page table.

2. The page tables return five fields which are loaded into the registers {*VNR, VRR, VTR, VSR, VAR*}.

3. The *VNR* register is compared with the object id and an object fault interrupt generated if they differ. This occurs whenever the hashing function selects a pager entry which corresponds to a different object. If an interrupt is generated, the current entry in the pager table is unloaded, and the correct entry is loaded in its place.

4. In parallel with step 3, the base address of the object, in the *VAR* register is added to an index register and sent to the dynamic ram.

5. In parallel with step 4, the index register is compared with the size register (*VSR*) and a fault signalled if addressing is out of bounds.

6. Two cycles later the DRAM bank that holds the heap delivers the addressed word of the object.

The steps 1. . . 5 can be completed in one clock cycle.

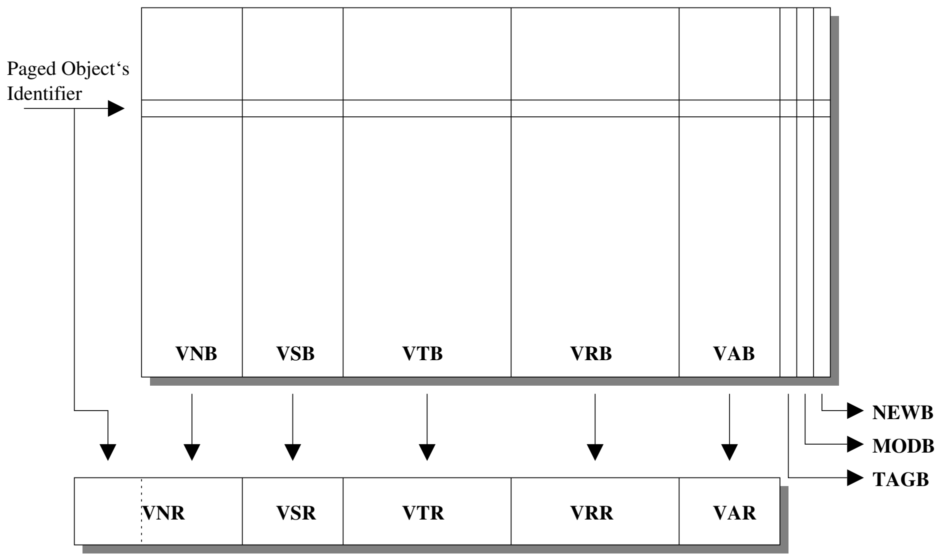

Figure 13: Rekursiv Pager Table Layout

This design is unusual in that it stores the type and first value field of an object in the page tables. Since these are implemented in high speed static RAM, it enables type information and the most frequently used value field to be available before the DRAM cycle has finished. Since the processor is designed to execute Smalltalk like languages which dispatch methods depending upon the object type, this could lead to considerable speed gains.

One consequence of the method used to index the page tables, is that no two objects with the same lower 16 bits of their OID can co-reside.

1.4.2 Object Allocation

Special auxiliary hardware is provided in the memory management unit to handle the allocation of objects. A pair of hardware registers hold the top free address on the heap and the last object number allocated. Data-paths are provided to update the page tables and appropriately increment these registers in a single operation when creating a new object.

If this results in heap overflow, a garbage collection fault is signalled and a micro-coded garbage collection routine entered. Tag bits in the page tables are used for the mark phase of the garbage collection. After marking, the heap is then compacted. Since all objects are addressed indirectly through the page tables, only these have to be altered during compaction.

1.4.3 Network Addressing

An object identifier contains 38 bits available for object addressing once the tag bits have been taken into account. The preferred use of these, according to Linn, is for six bits to be used to identify on which machine on a local net the object resides. The remaining 32 bits are a re-usable object number. When object numbers are exhausted, a global garbage collection is performed and objects are reallocated object numbers lying in a dense subset of the 32-bit range.

This poses problems. The reuse of object numbers is feasible on a single machine, but on a network it becomes impracticable. Suppose that machine A contains an object referred to by an object on machine B. If the object on machine A is deleted and its object number reused, then accesses from machine B will return the wrong object.

The Rekursiv provides a large object address space compared to the SYSTEM/38, 2^{38} objects for an isolated machine, 2^{32} objects for a machine on a network as against the 2^{24} objects for the IBM machine. This means that the problem of object number reuse is less severe. But against this must be weighed the sorts of languages that are run on the two machines. Languages that support garbage collection, such as those supported on Rekursiv are much more profligate in their use of object numbers. It is the opinion of the authors that the 40-bit object numbers of the Rekursiv would prove a restriction were the machine to go into general use.

1.5 Summary

Over the previous pages descriptions some of the main object oriented systems have been presented, and a number of points have been raised concerning the suitability of each in supporting the ideal object oriented architecture. The creators of these systems held a number of differing beliefs from the others, and therefore it is of no great surprise to find that features considered essential within one design were ignored within another.

In order to summarize there systems, eight points were considered (in no particular order):

- The *Network Size* which was reachable by the system in question was classified into *LAN* (Local Area Networks), *WAN* (Wide Area Networks), and *N/A* (*i.e.* no networking).
- Whether or not the system was a *Tagged Architecture* (*i.e.*supported self-identifying data structures).
- *Paged Memory* availability is often considered to be essential in supporting access to very large object sizes.
- The *Virtual Memory Size* is also important, as it gives an estimate as to how long the system can last without reusing object identifiers.
- *Segmented Memory*, *i.e.* the support of variable sized, hardware assisted memory segmentation, is frequently used in providing object protection and management mechanisms.
- The availability of general purpose *Garbage Collection* external to the compiled language is attractive in making object oriented architectures more secure than systems using other designs.
- The *Commercial Scale* of an implementation is of interest, as it shows the amount of industrial interest and support currently vested with the particular system (as well as its availability).
- All *Available Languages* for each system have also been shown, and this helps to show the flexibility of each architecture examined.

A table showing the four systems covered (including two entries for the two MONADS designs) against the points raised above can be found in Table 1.

Table 1: Summary of Object Oriented Architectures

	Machine Architectures				
	MONADS-PC	MONADS-MM	MUTABOR	REKURSIV	SYSTEM/38
Network Size	LAN	WAN	N/A	LAN	N/A
Tagged Architecture	No	No	No	Yes	Yes
Paged Memory	Yes	Yes	Yes	No	Yes
VM Size (bytes)[a]	2^{60}	2^{128}	2^{72}	2^{38}	2^{48}
Segmented Memory	Yes	Yes	Yes	Yes	Yes
Garbage Collection[b]	No	No	?[c]	Yes	No
Commercial Scale	Experimental	Experimental	Experimental	Small Scale	Large Scale
Available Languages	Pascal LEIBNIZ	Pascal LEIBNIZ	*Library Access*	C Lingo PS-Algol Prolog Smalltalk	RPG

[a] Calculated from its PID size and maximum offset for consistency to MONADS. The quoted value for MUTABOR is 2^{32}.
[b] In the future automatic garbage collection on MONADS may be implemented at the architectural level.
[c] Unknown, but garbage collection is predicted to be part of the language structure and therefore not automatic.

1.6 Conclusions

Although the oldest of the persistent architectures, the SYSTEM/38 shows features that subsequent efforts to produce persistent systems would do well to learn from: a large address space, well thought out support for operating systems and an efficient paging mechanism. It has been a considerable commercial success with tens of thousands of sites running the machines. Since it comes packaged with its own operating system and relational database which provide a high-level user interface, most users remain unaware of the novelty of its underlying architecture.

The more modern MONADS-PC and MM architectures represent an interesting branch from the traditional view of persistent object oriented systems, by using a block structure approach to the design. This does support the current block-like languages of today, but may limit the development of future (and perhaps even current) object oriented languages. Its module grouping structure, while simplifying many elements of program construction, reduces the overall flexibility of the system.

Additionally, extra complexity is added to the system by way of module interfacing and process control, and this complexity is made only too clear to the programmer [31]. However, the object mapping mechanisms used in MONADS represents many of the features which have been demonstrated by current research as highly desirable.

MUTABOR, the research project from GMD, demonstrates the effectiveness of large translation buffers within object oriented systems. Its fine-grained approach to object management is also attractive, in comparison to the more coarse-grained SYSTEM/38 and MONADS designs. In addition, the project also highlights the need for short object identifiers for active space objects, which is desirable in any limited data bus system (especially in systems build around non-custom parts, where the overhead of long object identifiers can be severe). On the negative side, MUTABOR is clearly a complicated architecture (perhaps needlessly so). The amount of table traversal required to locate an object which is not present within the object buffer can be high, and therefore the probability of memory faulting during the traversal is also high. Such a design may not be desirable for real-time architectures.

Finally, the Rekursiv was examined. Its design was such that many of the common object management functions were implemented in microcode, and although this had the effect of improving certain system functions, many other parts of the design were adversely affected by the processor complexity (partially created by the three-chip implementation). The limited virtual memory space was also a problem, especially when used on a network. The poor performance of the design, the difficulties in porting languages so as to best use the writable microcode store, and the networking and virtual memory limitations all assisted in preventing the Rekursiv from becoming desirable to the public.

2 An Ideal Persistent Object Machine

Considering the past experience in building persistent object machines, what are the lessons to be learned and what are the conclusions that ought to be drawn for future machines of this class?

First look at the requirements that might be place on a machine.

- It should support the abstraction of an infinite object store.
- It should support the abstraction of objects being of infinite size.
- It should support the single-level store abstraction to obtain independence of media and geographical location.
- The implementation mechanisms should be kept simple.
- It should be designed to make the secure implementation of operating systems possible. In particular the synthesis of object identifiers should be prevented.
- It should make the minimum of assumptions about the target languages that it will support.

2.1 Logical addressing mechanism

When dealing with computers, infinities are abstractions. In practice there are only finite physical stores, but from an architectural point of view an infinite store means two things. Firstly that the virtual address space should be so much larger than what is physically realizable, that the virtual address never appears as limit. Secondly, the size of the memory that appears infinite is linearly related to processor performance. Bell and Newell [32][page 46] quote Amdahl, who, during the design of the SYSTEM/360, estimated the ratio at about 1 megabyte of RAM per MIP. In dealing with the provision of physical RAM, this is still a reasonable lower limit. When considering the virtual address space of a persistent machine, a similar linear relationship will hold, but one would expect the constant factor to be much higher. In practice one can set an upper limit by requiring that a machine creating objects at its maximum rate should not exhaust its address space in its expected service life.

A machine creating objects at a rate of 100,000 a second for ten years would create about 10^{12} or 2^{40} objects. If one allows some factor for safety, allowing 2^{48} objects per machine is a reasonable minimum. If one were to allow objects to be identified by a combination of a unique network address such as an ethernet address and an object number one reaches a minimum address space of 2^{96} objects. This is the same as the MONADS massive memory machine. Other architectures fall some way short of this.

The maximum size of individual objects should also approach infinity, but in this case the criteria for infinity is related to the precision of arithmetic on the machine. An object should be able to contain the largest addressable array, which in turn, is related to the largest integer supported on the machine. Since the sizes of integers often vary across a range of machines, this implies that in an architectural specification it would be wise to keep the object ID and address within an object as distinct entities rather than collating them in a single larger entity: the global byte address. By maintaining their orthogonality, the possibility exists of a single persistent architecture spanning machines of differing arithmetic precisions.

2.2 Security

Contemporary persistent machines provide security by providing a restricted set of instructions to manipulate object identifiers. A drawback of this is that it imposes particular layouts on objects, which may contradict the requirements of programming languages. This is less severe on tagged machines like the SYSTEM/38 than on machines like MONADS which demand the total segregation of pointers; but it still exists. Ideally one would want object IDs to be secure, but to have no restrictions on where they were placed in data-structures like variant records.

If an object ID can be placed at any byte address and if any part of it can be freely modified, which is what the semantics of programming languages with union types imply, then security can only be cryptographic. This can be ensured by:

- Making the object address space sparse
- Making it expensive to predict valid object IDs given any known valid ID.

Sufficient sparseness of address-space implies that the chances of discovering valid IDs by random probing can be reduced to negligible levels. The impossibility of systematically deducing unknown but valid IDs from known valid ones can be ensured by combining with the systematically defined portion of the object ID a random bit-string derived from some thermal noise source. This approach has been advocated by Wallace [33] and Cockshott [34]. Since the only portion of the ID that must be systematic is the network address a possible format for an object identifier might be:

- 48-bit ether address of owner,
- 80-bit random number.

2.3 Address translation

A general principle in computing is to use caches wherever practicable. Nowhere is this more relevant than in translating addresses. A persistent store machine has available three levels of address translation caching:

1. *Inverted tables* in the form of inverted page tables have already been employed on MONADS and SYSTEM/38. In slightly modified form, similar software maintained tables are used in many software only persistent object systems.

2. *Translation Look-aside Buffers* have traditionally been used in virtual memory systems to avoid looking up the page tables on each memory access. Although this use involves modest address lengths, the technology scales well. Silicon area used grows linearly with address length and response time logarithmically with address length. It is quite feasible to design on-chip TLBs that will handle 128 bit object addresses.

3. *Segment register* caching as used in the Intel 386 [35] and IBM RS/6000 architectures. In these machines the segment registers are composed of a user-visible selector or key field and a hidden set of address mapping fields. When the programmer loads the key field, system tables are consulted to load the hidden fields with base-address, limit and type of the object. With the address information in registers, subsequent accesses to it are fast. It is the job of a compiler to minimize the number of segment register loads.

These mechanisms form a cascade, with the first loading the second which in turn loads the third. Their effect is to translate object addresses into what Intel call a *linear address space*. This space is only accessible to operating system level code. The linear address space can either be directly interpreted as physical RAM addresses, or, can be directed to a paging mechanism. In the first case, analogous to the Rekursiv, persistence would be established by the transfer of entire objects between volatile and non-volatile store. In the second case, transfers would occur at the page level. With a paged linear address space, the linear address length should be chosen to be sufficient to address all of the secondary store on the machine. With current technology that implies a linear address space of the order of 2^{40} bytes.

An advantage of separating the paging mechanism from the object addressing mechanism is that it becomes possible to support a range of machines at different performance levels, each of which might have a different size of linear address space. Such changes, occurring below the level of the object store, would be invisible to user software allowing the binary portability of code between machines at different performance levels.

3 Conclusion

Naturally, there is a significant difference between what is currently implemented and what has been highlighted as being attractive. This stems from differences of opinions as to what features of an object oriented system are attractive and in what order of merit they should be placed, and also as to what features can actual be implemented efficiently.

Within the four systems examined, only three points were consistently identified as desirable:

- The use of segmentation techniques to create variable-sized memory blocks within the main store.

- Protection of these segments from accidental and malicious modification and access.

- Providing the system with a sufficiently large address space such that, in the lifetime of the system, the addresses need never be reused.

Of these four, the MONADS-MM system comes the closest to the model highlighted above and within current literature. Note that even this implementation of the ideal is constrained by other factors; the actual development fund for this system was seriously limited, requiring much of the development work do be done on a voluntary basis. Perhaps future research into implementation techniques can help to minimize the deviations of next-generation object oriented architectures from the ideal system model.

References

[1] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer, "One-level storage system," *IRE Transactions*, vol. 2, pp. 223–235, Apr. 1962.

[2] T. Kilburn, D. B. G. Edwards, and G. E. Thomas, "A transistor digital computer with a magnetic drum store," *Proceedings of the IEE*, vol. 103B, no. sup 1-3, pp. 390–406, 1956.

[3] E. I. Organick, *The MULTICS System: An Examination of its Structure*. MIT Press, 1972.

[4] H. Whitfield and A. S. Wight, "EMAS - the edinburgh multi-access system," *Computer Journal*, vol. 16, no. 4, 1973.

[5] G. A. Blaauw and F. P. Brooks, "The structure of the SYSTEM/360," *IBM Systems Journal*, vol. 3, no. 2, pp. 119–135, 1964.

[6] SUN Microsystems, *SunOS System Services Overview*, Mar. 1990.

[7] J. K. Iliffe, *Basic Machine Principles*. London: MacDonald, 1968.

[8] J. L. Keedy, "An outline of the ICL 2900 series system architecture," *Australian Computer Journal*, vol. 9, pp. 53–62, July 1977.

[9] G. Soltis, "Design of a small business data processing system," *IEEE Computer*, pp. 77–93, Sept. 1977.

[10] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, "IBM SYSTEM/38 support for capability-based addressing," in *The Eighth Symposium on Computer Architecture*, pp. 341–48, May 1981. Published in SIGARCH Newsletter, Vol 9, No 3.

[11] D. Abramson, "Hardware management of a large virtual memory," in *Proceedings of the 4th Australian Computer Science Conference*, (Brisbane), pp. 1–13, 1981.

[12] D. B. G. Edwards, A. E. Knowles, and J. V. Woods, "The MU6-G. a newdesign to achieve mainframe performance from a mini sized computer," in *Proceedings of the 7th Annual Symposium on Computer Architecture*, pp. 161–167, 1980.

[13] A. Chang and M. F. Mergen, "801 storage architecture and programming," *ACM Transactions on Computer Systems*, vol. 6, Feb. 1988.

[14] A. Malhotra and S. J. Munroe, "Support for persistent objects: Two architectures," in *Proceedings of the Hawaii International Conference on System Sciences*, IEEE Press, 1992.

[15] J. Rosenberg and D. Abramson, "MONADS-PC - a capability-based workstation to support software engineering," in *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, pp. 222–231, 1985.

[16] D. Koch and J. Rosenberg, "A secure RISC-based architecture supporting data persistence," in *Computer Architecture to Support Security and Persistence of Information*, pp. (10–1)–(10–14), University of Bremen, May 1990.

[17] J. L. Keedy and J. Rosenberg, "Uniform support for collections of objects in a persistent environment," in *Proceedings of the Twenty Second Annual Hawaii International Conference on System Sciences* (B. D. Shriver, ed.), vol. 2, pp. 26–35, IEEE Computer Society, 1989.

[18] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *ACM Operating Systems Review*, vol. 13, no. 2, pp. 3–19, 1979.

[19] J. L. Keedy, "A comparison of two process structuring models," tech. rep., Monash University, 1980. MONADS Report Number 4.

[20] R. S. Fabry, "Capability-based addressing," *Communications of the ACM*, vol. 17, pp. 403–412, July 1974.

[21] J. L. Keedy, "An implementation of capabilities without a central mapping table," in *Proceedings of the Seventeenth Annual Hawaii International Conference on System Sciences*, pp. 180–185, 1984.

[22] J. Rosenberg, J. L. Keedy, and D. Abramson, "Addressing mechanisms for large virtual memories," tech. rep., St Andrews University, 1990. CS/90/2. To appear in the Computer Journal, Aug. 1992.

[23] J. Kaiser, "MUTABOR, a coprocessor supporting memory management in an object-oriented architecture," *IEEE Micro*, vol. 8, Oct. 1988.

[24] Y.-C. Liu, *The M68000 Microprocessor Family*. Prentice-Hall International, 1991.

[25] J. Kaiser and K. Czaja, "An architecture to support persistence in object-oriented systems." Available from the authors at kaiser@gmdzi.gmd.de or czaja@gmdzi.gbx.de, 1990.

[26] F. J. Pollack, K. C. Kahn, and R. M. Wilkinson, "The iMAX-432 object filing system," in *Proceedings of the Eighth Symposium on Operating System Principals*, vol. 15, pp. 137–147, SIGOPS, Dec. 1981.

[27] W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA/C.mmp: An Experimental System*. McGraw-Hill, 1981.

[28] G. T. Almes, *Garbage Collection in an Object-Oriented System*. PhD thesis, Carnegie-Mellon University, June 1980.

[29] J. Kaiser, "An object-oriented architecture to support system reliability and security," in *Computer Architecture to Support Security and Persistence of Information*, pp. (9–1)–(9–15), University of Bremen, May 1990.

[30] D. M. Harland, *REKURSIV, Object Oriented Computer Architecture*. Ellis Horwood Limited, 1988.

[31] J. Rosenberg, "Pascal/M - a pascal extention supporting orthogonal persistence," Tech. Rep. 89/1, Department of Electrical Engineering and Computer Science, the University of Newcastle, 1989.

[32] D. P. Seiwiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*. McGraw-Hill International, 1982.

[33] C. S. Wallace and R. D. Pose, "Charging in a secure environment," in *Security and Persistence*, Springer Verlag, 1990.

[34] W. P. Cockshott, "Design of POMP - a persistent object management system," in *Persistent Object Systems*, Springer Verlag, 1990.

[35] Intel, *386™SX Microprocessor Programmer's Reference Manual*, 1989. Published by McGraw-Hill.