An Object-Based Processor Cache

Gordon Russell Paul Shaw

August 20, 1993

ARCH-12-93

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

Abstract

In the past, many persistent object-oriented architecture designs have been based on traditional processor technologies. Such architectures invariantly attempt to insert an object-level abstraction mechanism over the traditional processor's virtual addressing scheme; this results in an architecture which incurs a translation overhead on every object access. Other architectures use objects at the instruction level, but then use a virtual-based caching scheme. This may require bounds-checking, and even object-to-virtual translation, to be performed on every object access.

A new architecture, DAIS, is proposed which utilizes objects in instructions and in the caches. This paper presents a short history of persistence, analyses a number of persistent architectures, and presents the DAIS design strategy. The object-based caching mechanism of DAIS is described, involving topics such as object protection via tag bits, object- and page-based locking, range checking, object to virtual mapping function, and use of a secondary descriptor cache. The cache design results in a processor which is no slower than conventional processors based on virtual memory. The design is then extensively analysed for performance with differing cache sizes. This analysis indicates that using a secondary descriptor cache can increase performance by 21% over a system with instruction and data caches alone.

1 Introduction

Research into persistent object-oriented architectures has been underway for three decades. Work on persistence is ultimately derived from the development by Kilburn *et al.* of the Manchester University Atlas machine [17], which introduced the idea of *virtual memory*. Virtual memory was initially used as a mechanism for supporting disk caching. In an earlier machine, Kilburn [18] combined the then new transistors with a magnetic drum for the instruction store. Previous to this, most stores were constructed from William's Tubes, mercury delay lines, magnetic cores, and moving magnetic devices. The drum-based instruction store resulted in a slow machine cycle time of 30 milliseconds, but with a store of significantly lower cost than its competitors. The virtual memory of the Atlas was used to improve the performance of drum-based systems; information was still held on the drum, but a small number of pages could be *cached* by re-mapping their virtual addresses onto magnetic cores.

At this point in history, computers were still used as processing machines utilizing off-line storage. This differs from the modern view of computers, which can interact with on-line storage and act as long-term data repositories. Even in the 1960s, with all main memory technologies based on magnetic effects and thus non-volatile, long-term data was always relegated to tapes.

The development of disk storage, which was viewed from the start as a long-term storage medium, offered interactive access to a persistent store. Initial attempts to map disks and short-term memory into a single-level store, *e.g.* Multics [20] and Emas [30], were not universally accepted. Here, the desire was to map each data and instruction file of the type previously stored on tape onto a range of virtual addresses. This was not without problems: disk storage requirements exceeded address-space limits, and if a range of addresses were allocated to a particular file, then file growth was difficult to perform without relocating the entire file.

As a solution to the problems introduced by the single-level store, the idea of *segmentation* was advocated by Iliffe [2]. With a segmented architecture, memory is segmented into variable length segments, each identified by a unique segment id. In many systems, segments are also referred to as *objects*.

In an object-oriented heap, no user-level virtually-addressed memory accesses are permitted. Instead, all data and instructions needed by applications are partitioned into objects. These objects are created with a specified length, and are identified using an *object identifier* (OID). An object can be referenced using the OID, and data elements within an object accessed using OID[index]. Using an index which is larger than the object length is illegal and should be trapped. Allowing users to generate descriptors independently should be carefully controlled, since this could allow users to find objects which they should have no access to.

2 Object-Oriented Architectures

Many programming languages, such as C, use linear addressing schemes as their underlying memory structure. Here, segmented architecture support would offer no significant advantages. However, object-based languages can make extensive use of an object-oriented heap, where the extra layer of indirection introduced between processor and memory allows for powerful system-level (as opposed to compiler-level) management of objects.

When using object-based addressing, it is important to verify that certain conditions hold when accessing objects via descriptors. Since it should not be possible for the user to fabricate descriptors (either by accidentally corrupting a descriptor or by deliberately trying to find objects created by other users), either descriptors must be immutable by user processes, or the probability of finding a valid object identifier without first being given it should be prohibitally small. If the user is able to modify descriptors, then the mapping process between descriptors and actual object addresses must be able to detect when a descriptor is illegal. On many systems, attempting to use a descriptor which does not correspond to an object in the heap results in the death of the current process.

Even if the descriptor is genuine, the index into the object must lie within the limits of the object length. Attempts to access an object outside its length should at least be signalled to the user. If the object heap uses multiple storage devices (*e.g.* RAM and disk), then the object being accessed must be present on a memory-addressable device (*e.g.* in RAM). This can be handled in a similar way to a virtual pager.

Supporting object addressing means that certain checks must be carried out at run time. Considering the frequency of heap-based data accesses in typical programs, software support for object accesses would dominate execution time. As such, object-access support can best be done using hardware. Additionally, with a data heap completely accessed using a system-wide object-based addressing scheme, system supported mechanisms for garbage collection and networking can be implemented transparently from the application writer.

Over the following pages, the object-address translation mechanism and object data caching of four object-based architectures are analysed; the IBM SYSTEM/38, MONADS, MUTABOR, and Rekursiv. These cover most aspects of current object-oriented designs. A more detailed analysis of object-oriented systems can be found in [24] and [25].

2.1 SYSTEM/38

The IBM SYSTEM/38 family of small business computer systems [28, 27] was announced in October 1978. The overall aim was to provide small-scale users with facilities previously available only on large systems. The architecture was designed to promote a long life-span, and cover a range of hardware implementations. Even today, elements of the SYSTEM/38 can still be found in the newer IBM AS/400 machine.

The instruction set used is independent of data types and hardware data-flow characteristics such as the number of physical registers [5]. The memory is constructed as a single-level store, and all memory management is performed automatically by the machine. Memory is accessed by objects using capabilities, which in turn identify objects using OIDs. A capability is a data record, which contains the OID of the object in question, access rights to that object (*e.g.* read only, read/write, *etc.*), and other system dependent information (*e.g.* object type) [11] [6]. Memory is configured as 40 bits wide; 32 data bits, 7 ECC (Error Correcting Code) bits, and a single tag bit.

Virtual-to-physical address translation is performed using an inverted page table. Hash collisions are stored in a linked list, and thus must be scanned serially. The entire collision chain for an object must be searched before an unmapped object can be detected as such, and this may have a detrimental effect on performance. If parts of the page table has itself been paged out to disk, then a failed search can also cause paging faults.

To further improve the translation performance, a hardware lookaside buffer is used (a TLB). This is organized as a 64 entry 2-way set associative cache. This gives a significant performance improvement [28]. A direct-mapped cache would have offered more opportunities for optimistic cache accesses, at the expense of a lower hit-rate.

SYSTEM/38 exhibits a large address space (although unable to support networked virtual addressing), well thought out operating system support, and an efficient paging strategy. It was a considerable commercial success, with tens of thousands of units sold. Packaged with its own operating system and relational database providing a high-level user interface, most users were completely unaware of the underlying object-based heap.

The lack of automatic garbage collection was an interesting omission from the design. The inclusion of collection techniques would have made the structure of the processor's object descriptor simpler to manage.

Unaligned memory transfers either complicates the off-chip memory subsystem and the cache, or incurs two memory cycles. Unaligned memory references are a rare occurrence in most programs [8][page E-12]. RISC philosophy suggests optimizing for the frequent case. Forcing alignment of capabilities to 16 byte boundaries would have allowed improved performance if the processor was scaled up to wider bus widths.

The type information which is part of each segment and object capability would best have been held within the objects themselves. Dynamic changes in types would be advantageous in 'hot' modification of executing applications, and this would need object relocation to another segment and global descriptor modifications. In SYSTEM/38, the complexity of updating every descriptor during dynamic re-typing could be high. The hierarchical segment/object structure imposed in SYSTEM/38 is not as flexible as a single-level object store.

Subroutines within the same process may have differing object access rights, and there are options to allow inheritance of access rights between subroutines (and also with child processes). Subroutine calls and returns can therefore be CPU-intensive tasks [10].

2.2 MONADS

The MONADS project was established in 1976. Its main objective was to examine and improve the methods used during the design and development phase of large-scale software systems. MONADS is still running today, and has branched out into the creation of an object-oriented environment based on hierarchical segmented virtual paged memory.

There have been four MONADS systems developed; MONADS-I, -II, -PC, and -MM. MONADS-I and -II were both based on Hewlett-Packard 2100A minicomputers, but in recent years the project team have been concentrating on MONADS-PC [15, 23] (a custom microcoded system) and -MM [19] (a custom co-processor and SPARC processor combination). Both machines can be programmed in a number of languages, including a dialect of PASCAL and (to a limited extent) LEIBNIZ [16]. MONADS was envisioned with three major design philosophies; a module structure, an in-process model for service requests, and a uniform virtual memory. A module may contain a number of objects and executable routines, but objects within a module can not be accessed from routines external to that module. Thus, objects contained within a module can only be accessed via routines within the object's module, with data being passed by value on a stack.

The module/object hierarchy of MONADS promotes low-overhead object management, both in terms of locking and networking [9]. However, this hierarchical design makes implementing orthogonality in object accesses difficult to achieve; given any OID, it is impossible to access that object unless the access is performed from within the surrounding module. While module-based objects efficiently support block-structured languages such as PASCAL and ADA, future (and even current) object-oriented languages may be performance limited. The module interfacing and process control adds extra complexity to the system, and this complexity is made all to clear to the programmer [22].

There is no garbage collection system for object, and when all objects within a module are no longer needed the module is explicitly deleted by the user. Even if garbage collection was introduced, objects cannot be deleted in isolation from its surrounding module. It may be possible for each module to contain routines which allocate and deallocate objects within the module's scope, with garbage collection to protect objects from illegal access after deletion and subsequent reuse. Collection would have to query every object within the transient closure of the root objects. For this, collection routines would be needed within each module in the heap.

2.3 MUTABOR

The Profemo project at Gesellschaft für Mathematik und Datenverarbeitung mbH[1] (GMD), while conducting research into the design of reliable distributed architectures, created the MUTABOR design. MUTABOR [12, 13, 14] (Mapping Unit for The Access By Object Reference) formed an object-oriented architecture supporting a secure and reliable computing base. Custom hardware was used to support both object and transaction management, coupled to a layered, generalized transaction-based operating system.

MUTABOR was constructed from a MC68020 processor, connected via its co-processor interface to a custom microprogrammable device. This increased the processor's instruction set to include object management functions, and permitted access to an address translation cache (ATC).

MUTABOR's philosophy is based on fine-grained objects, where for example a database of records would have every record mapped to a unique object. The same database implemented on MONADS would undoubtably contain all records within a single module. The MONADS module would possess interfaces to access a specified record, whereas each MUTABOR object (*i.e.* a single record) would have an associated type class and method interface.

Fine-grain objects mean that object-based locking is sufficient for transaction, locking, and network management. However, large objects are not well supported, since individual segments of an object cannot be locked in isolation of the whole.

The use of short object descriptors for data contained within the active space is unusual, but is a useful step in reducing the memory bandwidth and silicon overheads in handling long object descriptors. However, MUTABOR is an exceedingly complex machine (perhaps needlessly so), both in hardware terms and software interface.

[1]The German National Research Centre for Data Processing

2.4 Rekursiv

The Rekursiv chip-set (described in [7]), developed by Linn (a Glasgow phonography company), and fabricated by LSI Logic, was designed for use in the construction of object-oriented architectures. The original aim for the Rekursiv was to create a programming environment to allow support of production lines, where a processing unit would be placed in close proximity to each major component of the line. From this, the Rekursiv has been used in a variety of small-scale commercial and academic ventures.

The chip-set was made from three devices; the sequencer (LOGIK), arithmetic unit (NUMERIK), and the memory manager (OBJECT). LOGIK was microcode based, and its microcode store could be written to by the operating system, allowing instruction-based support to be implemented for whatever application (or language) was needed.

Rekursiv's virtual memory size is physically too small for many modern applications, and the heavily restricted network-address size made networking viable in only the smallest of networks. The ability to write microcode for the processor, and the object management functions which were also microcoded, did produce a performance improvement with respect to certain system functions. However, the added complexity of supporting this was detrimental to the performance of many other processor activities [4].

2.5 The Ideal Object Store

There is only one mistake that can be made in a computer design that is difficult to recover from — not providing enough addressing bits. . .

— *Bell and Strecker [1]*

The ideal object store should support the abstraction of an infinite size. In practice, only a finite object store is possible. However, a finite store can appear infinite provided that the virtual address space is much larger than can be used during the lifetime of the system. Consider a super-processor implementation of an object system. Give it 10ns main memory, 256 bit data bus, and a life-span of 100 years. In its lifetime, the processor could write and read (with no stall cycles) every location of a $2^{62.13}$ byte memory once. This would suggest that 2^{64} bytes of virtual memory for a single machine would be sufficient.

It is assumed that the average object size (including management overhead) for the object system is 512 bytes. Average object sizes (ignoring management overheads) given for MUTABOR (300 bytes) [14], iAPX-432 (300 bytes) [21], and Hydra (326 bytes) [31] suggest that this is a reasonable value. With a virtual memory size of 2^{64} bytes, this would allow 2^{55} objects to exist simultaneously upon a node.

For a networked machine, node addresses based on internet gives only 2^{32} possible combinations. This is slightly less than the number of people currently alive in the world, and is too restrictive if the addresses are geographically hierarchical. Instead, consider ethernet-based addresses, which use 48 bits.

On the network, objects must be fully identified, and therefore require at least 103 bits for identification. Networked object descriptors must be impossible (or at least prohibitively difficult) to forge, and some researchers (*e.g.* in [3] and [29]) have suggested adding a random number to networked descriptors, which acts as a descriptor *checkcode*. With a networked descriptor size of 128 bits, the probability of correctly forging a descriptor given a legal *local* descriptor is 34 million to one. If the local descriptors are randomly allocated, then forging an networked descriptor without knowing the local descriptor will be even more difficult.

Since most objects used by the local machine were created locally, the use of network-addresses and random-number bits is redundant for many descriptors. It is suggested that local object descriptors should be implemented using 64 bits, which can be aliased to 128 bit network object descriptors whenever a network object is being referenced. This has the benefit of halving the processor bandwidth requirements in handling descriptors, reducing the amount of space needed to hold a descriptor, and reducing the amount of processor transistors needed in handling descriptors. It also implements a useful level of abstraction between world-wide and local object naming conventions,allowing future modifications to the descriptor layouts. The network descriptor (named PIDs or *persistent identifiers*) to local descriptor (named OIDs or *object identifiers*) aliasing function can be controlled by an object management system.

The DAIS architecture is an attempt to solve the object caching and mapping problems identified during the systems' analysis. It makes use of a cache structure based directly on object descriptors and offsets, rather than the more common virtually or physically addressed cache. This cache structure has been designed to allow object bounds-checking to be performed by simple bit-ANDing, rather than arithmetic comparison.

2.6 Object-Based Heap

Object-based addressing is implemented using a three-layered approach, as depicted in figure 2. The top layer is the *object space*, where a 64 bit object identifier (OID) and a 32 bit index (unsigned integer) is used to access object data. Increasing the index by 1 moves on 8 bits within an object. An OID in object space is directly equivalent to the virtual address of an object descriptor in *virtual space*. The object descriptor contains the virtual address, object status, and the overall length (in bytes) of the object body (see figure 1). Accessing an object with an index greater than or equal to the object length causes a processor exception.

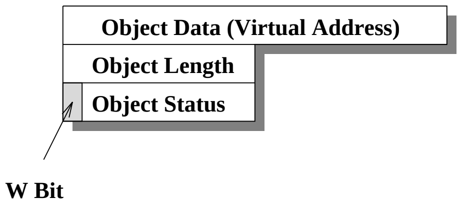

Figure 1: DAIS Object Descriptor

The *physical space* forms the third layer of DAIS' object-based heap. The virtual to physical mapping function can be controlled by an object management system to implement persistence and dynamic (*i.e.* run-time) object modifications which changes the internal structure of objects [24].

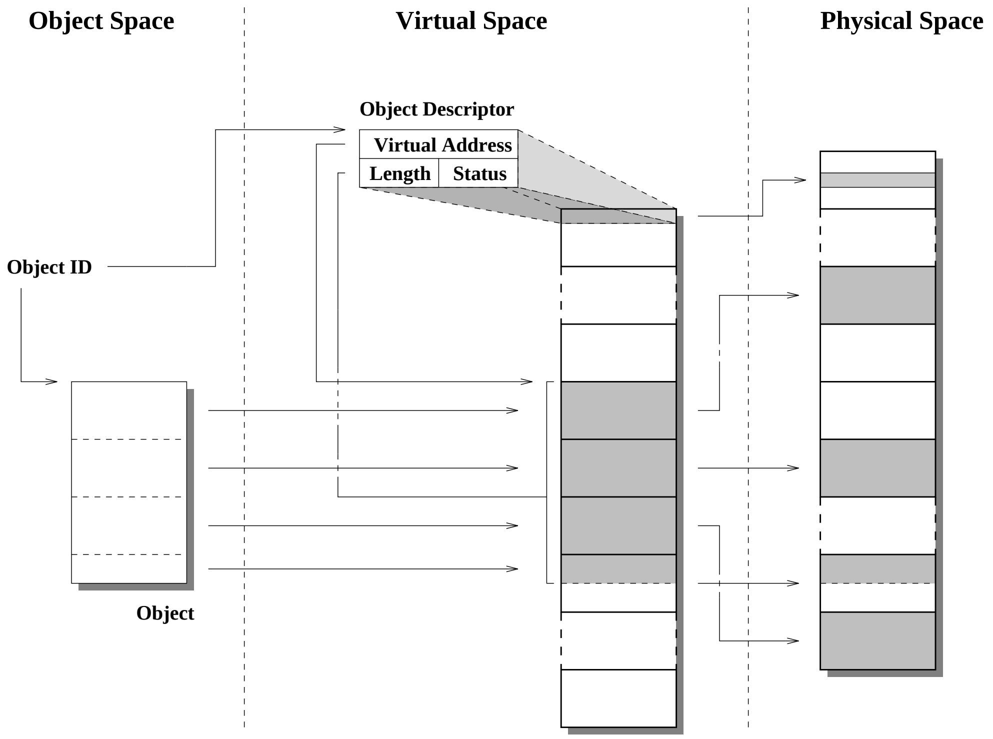

Figure 2: Mapping of Objects onto Physical Space

Object locking is designed to operate on two types of granularity; per object and per virtual page. The two different classes can be identified from the W bit, which is bit 0 of the object descriptor's status field. From this, and also from the current virtual protection which exists on the object-page being accessed, specifies the type of object locking:

Descriptor Page Protection	W Bit	Data Page Protection	Access Right
Page Unmapped	—	—	Object not accessible
Read or Read/Write	Set	Read or Read/Write	Object is read only
Read or Read/Write	Clear	Page Unmapped	Object page is not accessible
Read or Read/Write	Clear	Page Read Only	Object page read only
Read or Read/Write	Clear	Page Read/Write	Object page is read/write

Object space is accessed using a 65 bit data bus (which is also the size of a memory word), of which 1 bit is assigned to hold tag information. This tag is either set, indicating that the remaining 64 bits contains an object identifier, or clear, indicating that the remaining bits contains non-descriptor data.

Even though the word size is 65 bits, the 64 bits of data can all be accessed in 8, 16, 32, or 64 bit amounts (provided that the tag bit is clear) via various processor instructions. However, reading anything but the whole 64 bits of information causes a processor exception if the memory word in question contains an object descriptor (*i.e.* the tag bit is set).

2.7 Processor Cache

In general, object-based processors store a great deal of information on individual objects within the main object caches (*e.g.* object length, physical address, internal object layout details, *etc.*). DAIS holds much of this information within a secondary cache (called the *object cache* or O-cache), which is accessed only when the *primary* caches miss. DAIS uses two primary caches, one for data and one for instructions (the D-cache and I-cache respectively). The connections used between primary and secondary caches, and the stages involved in converting from object-based addressing to physical addressing is shown in figure 3.

Although the figure shows the CPU only operating with object-based addressing, it can also use physical addressing. Physical addressing is restricted to supervisor mode, and allows the object management routines to bypass both the fault manager and virtual mapper.

The CPU has two bus connections to external memory, both routed through the virtual memory manager. One bus is used during data and instruction cache misses, and the other is dedicated to register management (labelled *spill/refill path*. This second bus permits a register-file management system to perform register-memory transfers using virtual addresses (*e.g.* the spill/refill transfers incurred during window overflow/underflow on a window-based register implementation). Virtual addressing (as opposed to object addressing) is used to avoid contention between register-file management and data/instruction cache accesses.

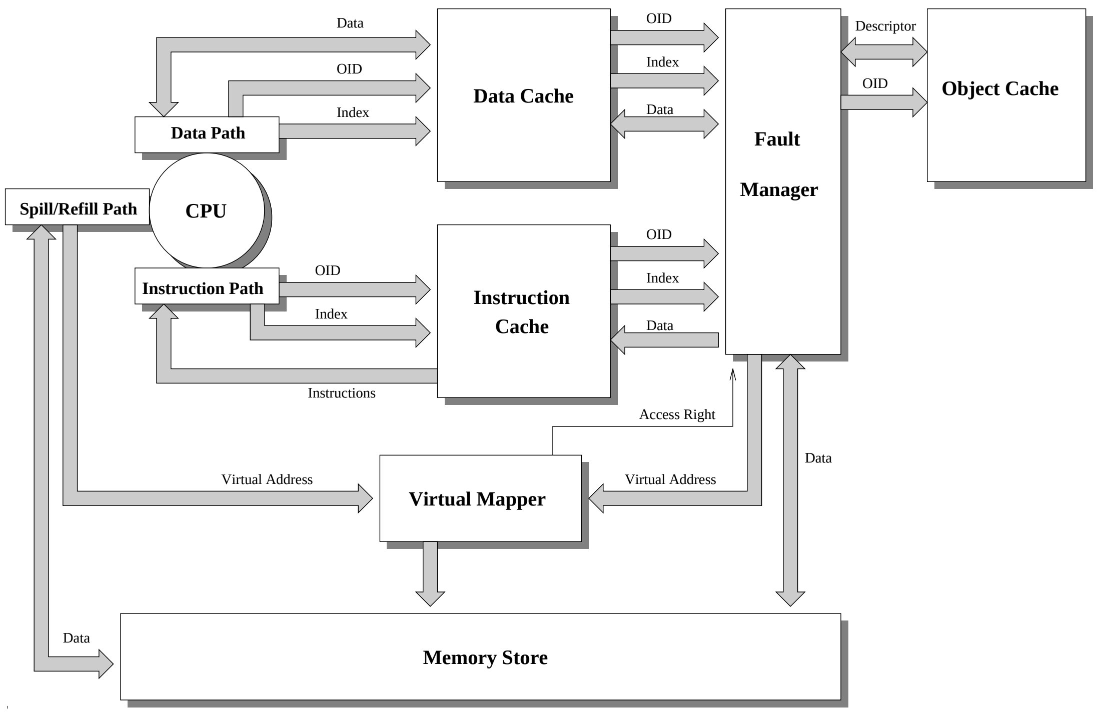

Figure 3: Overview of DAIS' CPU to Memory Pathways.

The primary data cache structure used in DAIS is shown in figure 4. The I-cache is similar in design, except that the hardware involved with writing to the cache is removed. The cache lines are direct mapped using a hash based on the desired OID and object index. A hit in the cache produces four pieces of information on the object; the cache line (*i.e.* the data stored at that point within the object), valid and dirty bits, and the S bit.

Each 8 bits of cache line is associated with a single valid bit. If set, then that portion of the cache line is valid, and vice-versa. Attempting to either read or write invalid portions of the cache produces an error (signalled on the *validity* line). These valid bits remove the need for subtraction based range-checking. The D-cache operates using a write-back policy, and thus the virtual address of an object is only needed on a cache-miss or cache-flush. Changing any part of a 64 bit (the maximum amount of data which can be written in a single memory cycle) segment of a cache line causes the corresponding dirty bit to become set. The S bit is used to indicate the access rights for that cache line. If the bit is set, then only reading of the line is permitted. With the bit clear, both reading and writing actions are permitted.

With so little information held within each cache entry, cache access rates should be similar to standard data caches. Additionally, since an object's virtual address and overall length changes infrequently (if ever), the primary caches are rarely flushed.

A primary cache miss is handled using a two-stage process, controlled by the *fault*

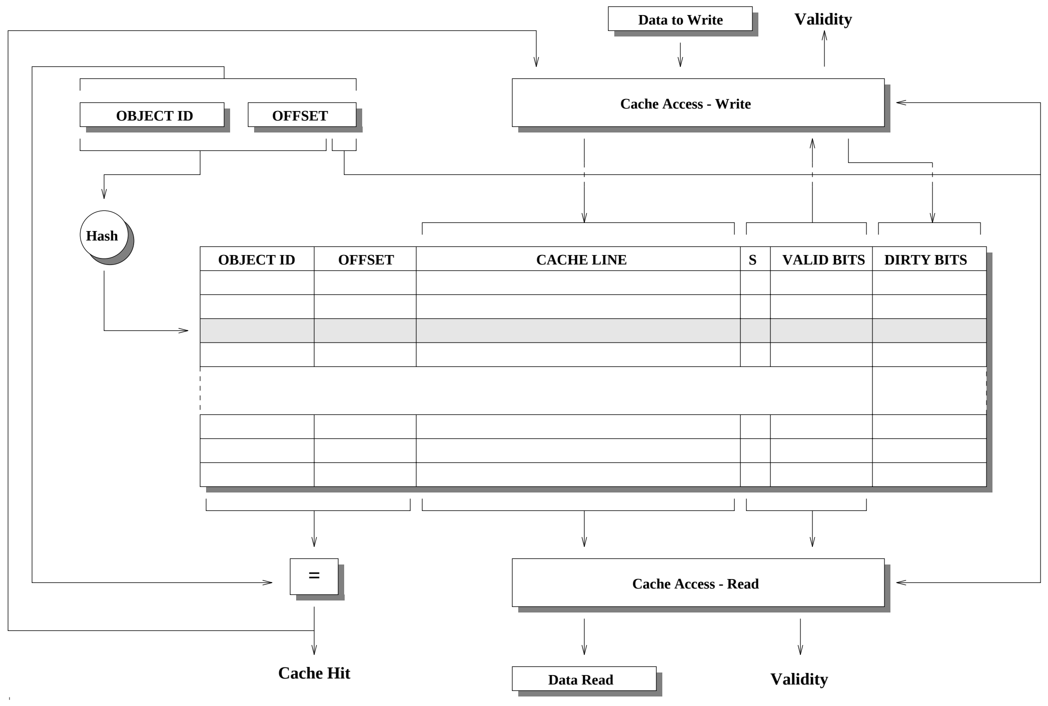

Figure 4: Structure of Data and Instruction Caches.

manager. Stage 1, performed only when the D-cache misses, examines the primary cache line to be overwritten by the new data. If the line contains modified data (detected using the dirty bits), then that data is written out to memory.

The fault manager calculates the address of the data which caused the cache miss, and stage 2 transfers that data from memory to the primary cache. It also calculates the valid bits for the cache line, setting a bit only when the corresponding 8 bits of data actually lies within the specified object. If the manager calculates that no valid bits would be set then the memory action is not performed, although the valid bits are still written back into the primary cache. The S bit is set in the primary cache if either the W bit is set in the object cache (*i.e.* bit 0 of the object status is set), or if during the data fetch for the primary cache the virtual manager reports read-only.

The address and length of objects accessed by the fault manager are obtained from the object cache. Should the object cache also miss, then the desired object information is first fetched from memory. Since the OID is also the virtual address of the object descriptor, no OID to virtual-address mapping function is required. The data paths between the object cache, fault manager, main memory, and the primary caches are shown in figure 5. Note that if any of the memory accesses cannot be performed (*i.e.* the virtual manager does not provide sufficient access rights to perform the desired action), then the current process is suspended until the situation is resolved (*e.g.* the object details are requested over the network).

In the current design specification of DAIS, the data cache contains 256 lines with a line length of 128 bits, and the instruction cache 256 lines of length 256 bits. This equates to a data cache consisting of 7584 bytes of storage, and the instruction cache to 12032 bytes. The object cache contains 128 entries, requiring 2464 bytes of storage. The total space for all three caches is approximately 21.56 KBytes, which is similar in size to many modern processor designs.

3 Benefits of the Object Cache

The layered approach to DAIS' caching structure has been put forward as a benefit over other object-oriented architectures. However, what exact benefit does the secondary (*i.e.* object) cache give? Here, a theoretical analysis of secondary cache performance is presented. The symbols used in this analysis are shown below:

M_I, M_I, M_I	$=$	I-, D- and O-cache miss rates
P_I, P_D, P_O	$=$	I-, D- and O-cache miss penalty
T_I	$=$	Time to read one I-cache line
T_D	$=$	Time to read/write one D-cache line
m	$=$	Fraction of instructions referencing D-cache
d	$=$	Probability that a particular memory word in D-cache is dirty
n	$=$	Number of memory words per D-cache line

To analyse the cycle-per-instruction (CPI) benefits of the O-cache, the analysis shall begin in general terms. To allow fair comparison with other RISC-based architectures, DAIS is considered to contain a pipeline which (ignoring pipeline stalls) allows single-cycle instruction execution. The overhead per instruction for I-cache misses is $M_I P_I$ cycles. Similarly, the overhead per instruction owing to D-cache misses is $mM_D P_D$. The total CPI is thus:

$$CPI = 1 + M_I P_I + mM_D P_D \tag{1}$$

Here, both miss penalties include the time taken to fetch the object descriptor and then fetch the cache line of object data. To discover performance with and without O-cache, it is necessary to derive formulae for P_I and P_D under both conditions.

3.1 CPI without O-cache

If the I-cache misses, then the miss penalty P_I must be the time P_O to load the object descriptor plus the time T_I to load in the missing cache line. On a D-cache miss, each dirty memory

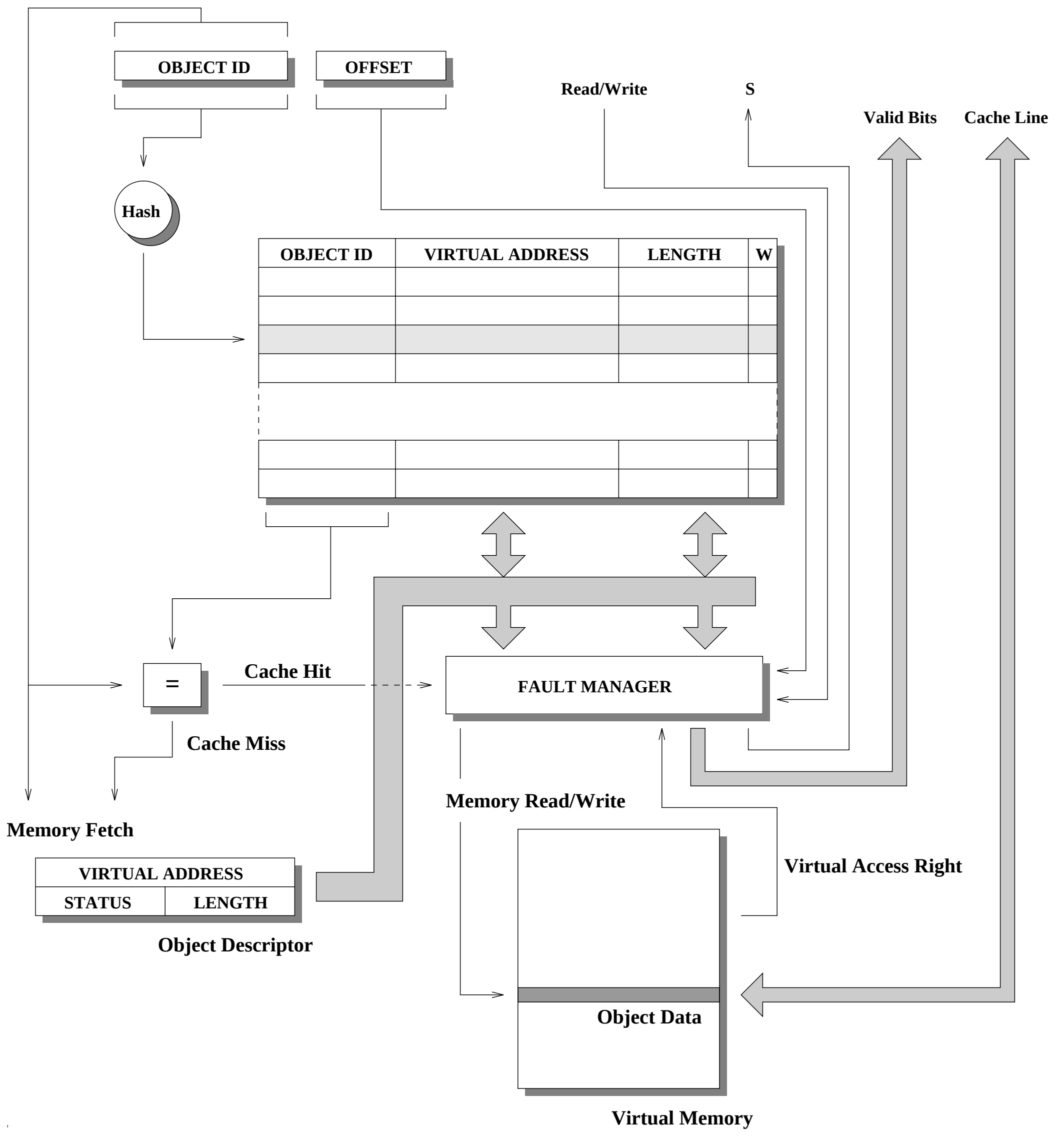

Figure 5: Structure of Secondary Object Cache.

word held in the cache line must first be written to memory. Given the probability d of a cached memory word being dirty, then P_I and P_D are defined as:

$$P_I = P_O + T_I$$

$$P_D = (2 - (1-d)^n)P_O + (1+d)T_D$$

3.2 CPI with the O-cache

When an object cache is included, the I- and D-cache miss penalties P_I and P_D must be updated to reflect O-cache hits and misses. On an I- or D-cache miss, the object cache is scanned. The object descriptor is returned in one cycle if contained within the object cache. If not, the object descriptor must be fetched from memory. In addition to scanning the object cache, the appropriate line of the cache must be read in.

P_D is again similar to P_I, except that is must also include the probability that the data line to be replaced contains dirty data. Saving dirty data requires access to the O-cache, which may in turn cause an O-cache miss. Thus:

$$P_I = 1 + M_O P_O + T_I$$

$$P_D = 2 + M_O P_O \left[2 - (1-d)^n\right] - (1-d)^n + (1+d)T_D$$

3.3 Some Example Figures

Here, some figures will be substituted into both CPI equations to give the reader some idea of the increase in efficiency the O-cache gives.

TeX, DeTeX, Zoo, and Fig2dev were simulated. The miss rates for direct mapped caches of 4K and 8K bytes with line lengths of 16 and 32 bytes are shown in tables 1 and 2. These were derived using SHADOW on a SPARC. For the case of the D-cache, the miss rates are *per instruction*, and not per D-cache access. This means that m can be omitted from equation 1 when estimating performance. The averages shown in these tables are both ‘straight’, and weighted on the number of instructions executed by each of the simulations. Other results from these simulations (pertaining to write-backs of cache lines) where used to estimate d at 0.2.

Unfortunately, the miss rate of the O-cache cannot be predicted exactly from such a cache trace. Analysis shows that over half the O-cache accesses are caused by I-cache misses. Since temporal locality of instruction objects is high, it is anticipated that nearly all O-cache misses will result from D-cache misses. For analysis purposes, the O-cache miss rate was assumed

Table 1: Data Cache Miss Rates (Per Instruction)

Benchmark	4K cache		8K cache	
	16 byte lines	32 byte lines	16 byte lines	32 byte lines
TEX	1.79%	1.88%	1.24%	1.23%
DeTEX	1.15%	1.93%	0.56%	1.32%
Zoo	2.46%	2.26%	1.84%	1.53%
Fig2dev	1.04%	0.99%	0.53%	0.47%
Average	1.61%	1.76%	1.04%	1.14%
Weighted Av	1.76%	1.86%	1.21%	1.22%

Table 2: Instruction Cache Miss Rates

Benchmark	4K cache		8K cache	
	16 byte lines	32 byte lines	16 byte lines	32 byte lines
TEX	5.99%	3.74%	3.74%	2.35%
DeTEX	7.31%	5.08%	3.73%	2.72%
Zoo	1.12%	0.78%	0.35%	0.23%
Fig2dev	9.95%	6.70%	5.74%	3.98%
Average	6.09%	4.08%	3.39%	2.32%
Weighted Av	5.94%	3.81%	3.74%	2.37%

to be 10%. Interestingly, doubling this value only has a small effect on the performance. In a system without an external cache, the performance degradation is under 4%. With external cache, the degradation is only 2%.

Four scenarios have been staged; all combinations of processors either with or without O-cache and an external cache. When the processor has no O-cache, each object descriptor must be fetched from main store. The assumption is made that without external cache each main store access requires 5 processor cycles. For a system with this external cache, access is assumed to require and average of 2 cycles. Together with the assumption of a 65 bit external data bus, these times determine P_O, T_I, and T_D. The miss rates used for the D- and I-caches where the *weighted averages* shown in tables 1 and 2. For all four cases, the resultant CPI (disregarding other processor stalls) is shown for different combinations of I- and D-cache sizes (4 and 8 KBytes) and line lengths (16 and 32 bytes) in figures 3–6. The minimum CPI from each cache size combination is highlighted.

Table 3: CPI with no O- or E-cache

		Instruction Cache			
		4K/16	4K/32	8K/16	8K/32
Data Cache	4K/16	2.64	**2.59**	2.20	**2.16**
	4K/32	2.93	2.89	2.49	2.45
	8K/16	2.50	**2.45**	2.06	**2.02**
	8K/32	2.67	2.63	2.23	2.20

Table 4: CPI with O- but no E-cache

		Instruction Cache			
		4K/16	4K/32	8K/16	8K/32
Data Cache	4K/16	**1.97**	2.10	**1.71**	1.78
	4K/32	2.22	2.34	1.95	2.03
	8K/16	**1.89**	2.02	**1.63**	1.70
	8K/32	2.04	2.17	1.78	1.85

Table 5: CPI with E- but no O-cache

		Instruction Cache			
		4K/16	4K/32	8K/16	8K/32
Data Cache	4K/16	1.66	**1.64**	1.48	**1.46**
	4K/32	1.75	1.74	1.58	1.56
	8K/16	1.60	**1.58**	1.42	**1.41**
	8K/32	1.66	1.64	1.48	1.47

Table 6: CPI with both O- and E-cache

		Instruction Cache			
		4K/16	4K/32	8K/16	8K/32
Data Cache	4K/16	**1.44**	1.48	**1.32**	1.34
	4K/32	1.54	1.58	1.42	1.44
	8K/16	**1.40**	1.44	**1.28**	1.30
	8K/32	1.47	1.51	1.35	1.37

From these four tables, the trade-offs in choosing D- and I-cache sizes and line lengths can be investigated. The benefits of including an object or external cache are also part of these tables. In all cases adding an O-cache resulted in better performance than doubling the size of a 4 KByte D-cache. For the cache setup detailed in section 2.7, the CPI are:

O-cache	External Cache	CPI
No	No	2.16
No	Yes	1.46
Yes	No	1.78
Yes	Yes	1.34

Hence, the O-cache gives a 21% speed improvement when there is no external cache present, and a 9% improvement if there is.

4 Conclusions

This paper has briefly reviewed current persistent architectures to ascertain inefficiencies inherent in them. Many of these can be removed by including objects at the instruction level and using object-based caching mechanisms.

DAIS operates on 64-bit object identifiers (plus one tag bit). These are then aliased to larger identifiers when worldwide communication is required. This has the advantages of halving the processor bandwidth requirements for descriptors, reducing space required to hold descriptors, and reducing processor transistors required for handling descriptors (smaller cache tags, registers *etc.*). An additional advantage is that world-wide and local object identification methods are separated, allowing future modifications to descriptor layouts.

The caching structure of the DAIS architecture has been presented incorporating features allowing the architecture to run at around the same speed as non object-based processors. DAIS achieves efficiency by providing only the minimum of support for objects. This boils down to one tag bit and an object descriptor including object size and status. On a RISC architecture, only load and store instructions need be concerned with objects, which greatly simplifies processor design.

A cache structure based on objects and offsets rather than on virtual addresses allows object data to be accessed without the need for an address translation. Such a scheme also allows bounds checking to be achieved simply by examination of validity bits, removing the need for arithmetic comparison. A bit in the status word of an object allows for locking at the object level. This is useful for small objects. For large objects, page-based locking is more desirable, provided through the virtual memory manager. DAIS combines both these types and holds the read/write information in the data cache, providing an efficient mechanism for deciding on the validity of writes.

Analysis has been undertaken on the effects of choosing different cache sizes. This was done by dynamically tracing binaries for four different benchmarks on a non object-oriented architecture. Such an analysis is sufficient to provide performance evaluations for instruction and data caches, but not for the secondary object cache. Therefore, the sole conservative assumption was made that a 128 entry object cache have a 90% hit rate. However we believe that the real figure will be higher that this. The results of the analysis showed that a non-superscaler version of DAIS with external cache takes 1.34 cycles to execute each instruction.

This result is for a system with 8K of instruction cache, 4K of data cache and 2K of object cache. The object cache justifies its existence since a system with 8K of both instruction and data cache but no object cache has poorer performance.

References

[1] G. Bell and W. D. Strecker. Computer structures: What have we learned from the PDP-11? In *Proceedings of the Third Annual Symposium on Computer Architecture*, January 1976.

[2] G. A. Blaauw and F. P. Brooks. The structure of the SYSTEM/360. *IBM Systems Journal*, 3(2):119–135, 1964.

[3] W. P[aul] Cockshott. Design of POMP - a persistent object management system. In *Persistent Object Systems*. Springer Verlag, 1990.

[4] [William] Paul Cockshott. Performance evaluation of the Rekursiv object oriented computer. In Veljko Milutinovic and Bruce D. Shriver, editors, *Proceedings of the Hawaii International Conference on System Sciences*, pages 730–736. IEEE Computer Society Press, January 1992.

[5] S. H. Dahlby, G. G. Henry, D. N. Reynolds, and P. T. Taylor. The IBM SYSTEM/38: A high-level machine. In Seiwiorek et al. [26], pages 533–536. Reprinted from IBM SYSTEM/38 Technical Developments, pp. 47–50, 1978.

[6] R. S. Farby. Capability-based addressing. *Communications of the ACM*, 7(17), July 1974.

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

[8] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 1990.

[9] Frans A. Henskens, John Rosenberg, and Michael R. Hannaford. Stability in a network of MONADS-PC computers. In *Computer Architecture to Support Security and Persistence of Information*, pages (16–1)–(16–10). University of Bremen, May 1990.

[10] R. L. Hoffman and F. G. Soltis. The IBM SYSTEM/38: Hardware organization of the SYSTEM/38. In Seiwiorek et al. [26], pages 544–546. Reprinted from IBM SYSTEM/38 Technical Developments, pp. 19–21, 1978.

[11] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM SYSTEM/38 support for capability-based addressing. In *The Eighth Symposium on Computer Architecture*, pages 341–48, May 1981. Published in SIGARCH Newsletter, Vol 9, No 3.

[12] Jörg Kaiser. MUTABOR, a coprocessor supporting memory management in an object-oriented architecture. *IEEE Micro*, 8(5), October 1988.

[13] Jörg Kaiser. An object-oriented architecture to support system reliability and security. In *Computer Architecture to Support Security and Persistence of Information*, pages (9–1)–(9–15). University of Bremen, May 1990.

[14] Jörg Kaiser and Karol 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.

[15] J[ames] Leslie Keedy. An implementation of capabilities without a central mapping table. In *Proceedings of the Seventeenth Annual Hawaii International Conference on System Sciences*, pages 180–185, 1984.

[16] J[ames] Leslie Keedy and John Rosenberg. Uniform support for collections of objects in a persistent environment. In Bruce D. Shriver, editor, *Proceedings of the Twenty Second Annual Hawaii International Conference on System Sciences*, volume 2, pages 26–35. IEEE Computer Society, 1989.

[17] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer. One-level storage system. *IRE Transactions*, 2(EC-11):223–235, April 1962.

[18] T. Kilburn, D. B. G. Edwards, and G. E. Thomas. A transistor digital computer with a magnetic drum store. *Proceedings of the IEE*, 103B(sup 1-3):390–406, 1956.

[19] David Koch and John Rosenberg. A secure RISC-based architecture supporting data persistence. In *Computer Architecture to Support Security and Persistence of Information*, pages (10–1)–(10–14). University of Bremen, May 1990.

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

[21] Fred J. Pollack, Kevin C. Kahn, and Roy M. Wilkinson. The iMAX-432 object filing system. In *Proceedings of the Eighth Symposium on Operating System Principals*, volume 15, pages 137–147. SIGOPS, December 1981.

[22] J[ohn] Rosenberg. Pascal/M - a pascal extention supporting orthogonal persistence. Technical Report 89/1, Department of Electrical Engineering and Computer Science, the University of Newcastle, 1989.

[23] John Rosenberg and David Abramson. MONADS-PC - a capability-based workstation to support software engineering. In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, pages 222–231, 1985.

[24] Gordon Russell. DOLPHIN*: Persistent, Object-oriented, and Networked*. PhD thesis, University of Strathclyde, 1993. Under Preperation. A draft copy is available by email from the author (gor@cs.strath.ac.uk).

[25] Gordon Russell and [William] Paul Cockshott. Architectures for persistence. *Microprocessors and Microsystems*, 17(3):117–130, April 1993.

[26] Daniel P. Seiwiorek, C. Gordon Bell, and Allen Newell, editors. *Computer Structures: Principles and Examples*. McGraw-Hill International, 1982.

[27] Frank G. Soltis and Roy L. Hoffman. Design considerations for the IBM SYSTEM/38. In Gerald E. Peterson, editor, *Tutorial: Object-Oriented Computing*, volume 2. IEEE Computer Society, 1987.

[28] G. Soltis. Design of a small business data processing system. *IEEE Computer*, pages 77–93, September 1977.

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

[30] H. Whitfield and A. S. Wight. EMAS - the edinburgh multi-access system. *Computer Journal*, 16(4), 1973.

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