Proc. IEEE 11th Int'l Conference on Data Engineering, March 1995, pages 35-43

A High Performance Configurable Storage Manager

Alexandros Biliris and Euthimios Panagos
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974
{biliris, thimios}@research.att.com

Abstract

This paper presents the architecture of $\mathcal{B}$eSS – a high performance configurable database storage manager providing key facilities for the fast development of object-oriented, relational, or home-grown database management systems. $\mathcal{B}$eSS is based on a multi-client multi-server architecture offering distributed transaction management facilities and extensible support for persistence. We present some novel aspects of the $\mathcal{B}$eSS architecture, including a fast object reference technique that allows re-organization of databases without affecting existing references and two operation modes that an application running on a client or server machine can use to interact with the storage system – copy on access and shared memory.

1 Introduction

This paper presents the design and implementation aspects of $\mathcal{B}$eSS, a $\mathcal{B}$ell Laboratories $\mathcal{S}$torage $\mathcal{S}$ystem, which is a storage manager that facilitates the development of high-performance database management systems. $\mathcal{B}$eSS's overall goal is to provide fast and transparent access to persistent objects, independent of their size and their physical location, in a distributed computing environment based on a client-server architecture.

$\mathcal{B}$eSS allows application programs to access and manipulate persistent objects directly on the segment on which they reside, without incurring in-memory copying cost. It employs a fast object reference mechanism that is based on memory mapping [19, 30, 34]. Our scheme, however, does not involve a greedy allocation of virtual memory addresses. Large objects spanning multiple pages can be accessed and updated transparently as if they were small objects. Efficient byte range operations such as insert, append, and truncate are also provided for those objects.

$\mathcal{B}$eSS prevents database corruption caused by bad pointers by storing control structures separately from data. The control structures are protected by ordinary mechanisms provided by the virtual memory management hardware.

The $\mathcal{B}$eSS architecture has been designed to be highly modular and extensible so that it can meet the diverse performance requirements of applications with different needs. Client applications may associate user-defined hook functions to be executed when certain primitive events occur. $\mathcal{B}$eSS traps primitive events as they occur and causes the associated hooks to be executed. This mechanism allows users to enhance and even modify the behavior of $\mathcal{B}$eSS and their applications without changing the application code or the internals of the $\mathcal{B}$eSS system.

Furthermore, the $\mathcal{B}$eSS server is intended to be an *open* server, capable of supporting a wide range of applications. Sophisticated users can link with the $\mathcal{B}$eSS server a trusted piece of code in order to build specialized servers, like SQL and multimedia servers.

$\mathcal{B}$eSS offers concurrency control via locking and recovery via logging to support the traditional ACID transaction properties. Data and locks accessed by a transaction remain cached on the client where this transaction was executed and can be used by subsequent transactions running on the same machine. Cache consistency is guaranteed by following the callback locking algorithm [17, 19].

Applications running on a client workstation usually have different performance requirements. A novel aspect of the $\mathcal{B}$eSS architecture is that it offers a number of different operation modes for accessing persistent data. These modes allow applications to take advantage of current advances in hardware technology such as shared memory multiprocessors and virtual memory management hardware offering a huge address space. In this paper we describe the following modes:

- *Copy On Access*. Applications operate on objects after copying them into a private buffer pool.
- *Shared Memory*. Applications operate on objects present in a buffer pool that is shared by all applications running on the same machine.

The architecture of the $\mathcal{B}$eSS storage manager is not tailored to a specific data model or language. It is possible to build relational and object-oriented database systems and persistent languages on top of $\mathcal{B}$eSS as well as home-grown specialized database systems. For example, the Ode object-oriented database system [1], built on top of EOS [8], will be using $\mathcal{B}$eSS. $\mathcal{B}$eSS is also being used as the storage engine of the AT&T's Prospector [18], a content based multimedia system, that requires an extended relational interface to $\mathcal{B}$eSS. Calico, a storage system for continuous media which is currently being implemented in AT&T Bell Labs [6], uses $\mathcal{B}$eSS to provide storage and transactional support for the metadata of continuous media as well as disk space management facilities.

The rest of the paper is organized as follows. Section 2 provides the storage structures used in $\mathcal{B}$eSS, pointer dereference, and issues related to preventing database corruption caused by bad pointers, detecting updates on objects, and extensibility. In Section 3 we present the multi-client multi-server $\mathcal{B}$eSS architecture. Section 4 presents the operation modes available to client applications. In Section 5 we compare $\mathcal{B}$eSS with related work and Section 6 concludes our presentation.

2 The $\mathcal{B}$eSS System Architecture

At the conceptual level, $\mathcal{B}$eSS manipulates *databases* that are collections of $\mathcal{B}$eSS *files*. $\mathcal{B}$eSS files contain *object segments* in which objects are stored. An object segment is the clustering facility provided to users to indicate that some objects need to be collocated. A $\mathcal{B}$eSS file groups objects so that they could be retrieved later on via a cursor mechanism. However, an individual object can be accessed directly without first accessing the file containing it.

At the physical level, the database consists of a number of *storage areas*, which are UNIX files or disk raw partitions. Storage areas, are partitioned into a number of *extents*, and allocation of disk segments from one of these extents is based on the binary buddy system, as described in [3]. Storage areas that correspond to UNIX files may expand in size by one extent at a time.

All objects in a $\mathcal{B}$eSS file are stored on object segments allocated from one storage area; however, a storage area may contain objects belonging to multiple $\mathcal{B}$eSS files. Initially all objects of all files in a database may be placed in the same storage area. To accommodate growth, objects within a $\mathcal{B}$eSS file can be moved to another storage area, and as we shall see in the next section, without affecting existing object references.

Since all objects in a $\mathcal{B}$eSS file are stored within a single area, the size of an individual $\mathcal{B}$eSS file is limited by the addressability of the operating system (typically, 2 GB). $\mathcal{B}$eSS offers *multifiles*, which behave like regular $\mathcal{B}$eSS files (e.g., they can be scanned, and objects can be created in them) except that they expand over multiple physical storage areas and therefore their sizes are not limited by the operating system. In addition, when a multifile expands over different physical devices, perhaps on different disk controllers, it provides a convenient mechanism for parallel I/O processing such as a parallel file scan. This capability is used extensively in Prospector and MoonBase systems to perform, on multiprocessor machines, fast content-analysis and indexing on large databases of multimedia objects [18].

2.1 Segment and Object Structures

Figure 1 illustrates the structure of an object segment. Each object segment consists of two basic parts: the *slotted segment* and the *data segment*, each of which is a sequence of one or more contiguous pages. The slotted segment contains a fixed-size header and an array of slots. The data segment contains the actual objects which can vary in size and consume no predetermined amount of space. If needed, an *overflow segment* is used to hold additional control information such as large object descriptors.

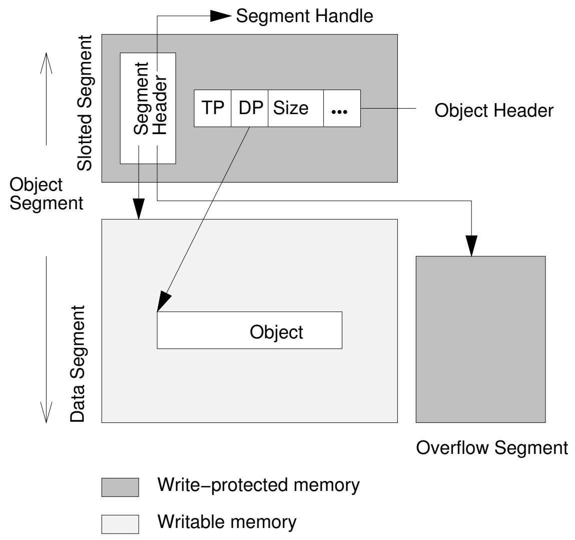

Figure 1: Segment and object structure.

For every object in the data segment, there is an object header that is stored in a slot in the slotted segment. The object header contains certain amount of meta-information that is necessary for managing the object it refers to – such as a pointer to the object's type (TP), a pointer to the object's data in the data segment (DP), the object size, and other bookkeeping information. Type descriptors contain the offsets of pointers within the objects they describe.

The slotted segment header includes information required for managing the object segment, such as the number of objects and the available free space in the data segment. It also includes a pointer to a memory control structure (referred to as *segment handle*) that keeps run time control information about the segment such as dirty pages, lock data, etc.; a pointer to the data segment; and a pointer to an overflow segment, if there is one.

The object identifier (OID) is a 96-bit number that uniquely identifies an object in a $\mathcal{B}$eSS system. It contains the host machine number, the database number, the offset of the object's header within the database, and a number to approximate unique oids – this number is stored in every slot and it is modified every time the slot is re-used. Slotted segments (and their slots) are allocated from one storage area and they are never relocated. Data segments can be re-sized or moved to a different location without affecting the validity of existing OIDs. It's possible that huge databases may end up having one storage area exclusively for slotted segments while data segments are spread over multiple storage areas.

However, object references are not implemented through the rather long and slow OIDs. Instead, references among objects belonging to the same database are pointers to the headers (slots) of the referenced objects. References among objects belonging to different databases, are implemented via a level of indirection: the reference points to a *forward* object, which is stored in the database of the referencing object, that contains the complete address of the referenced object. Such inter-database references are handled transparently and exclusively by $\mathcal{B}$eSS. After intra- or inter-database references are processed as discussed in the next paragraph, they point to the header of the referenced object, which in turn contains in its DP field the virtual memory address of data part of the referenced object.

To illustrate how the above structures are used, consider the actions taken when a *slotted segment fault* occurs. First, the slotted segment is fetched in memory, and a virtual memory address range for its data segment is reserved and access-protected. Then, for every slot in the slotted segment representing an object in the data segment, the DP field of the slot – whose value is the address in which the object was mapped the last time it was accessed – is adjusted to point to the new (reserved) virtual memory address of the object in the data segment. This involves just two arithmetic operations.

Suppose now a *data segment fault* occurs when some object within a data segment, for which addresses have been reserved as discussed above, needs to be accessed. First, the data segment of the object is either fetched as a whole or only the pieces needed to access the object are fetched – this depends on the availability of cache space. Second, $\mathcal{B}$eSS examines the type descriptor of every object O that is present in the portion of the data segment that was actually fetched and locates all references to other objects. For each reference to an object O_i, $\mathcal{B}$eSS performs the following actions. If the slotted segment containing O_i has never been referenced before in the current transaction, an address range for the slotted segment is reserved and access-protected. (If it has been referenced before, the slotted segment is either in memory or virtual memory addresses have been reserved for it.) Then, O is modified to point to the virtual memory address of the header of O_i. When later on O_i is accessed, there will be a slotted segment fault which will trigger the actions described in the previous paragraph.

Thus, accessing an object potentially causes actions in three waves. In the first wave, address ranges for the referenced slotted segments are reserved. In the second wave, as some of these slotted segments are accessed for the first time, slotted segments are fetched in and address ranges for the corresponding data segments are reserved. Finally, accessing some objects within one of these data segments causes the data segments to be fetched. The latter may trigger another round of virtual memory address reservation and data fetch.

Regarding large objects, $\mathcal{B}$eSS offers transparent access to fixed-size objects that cannot fit in a data segment – currently, up to 64KB. For such a large object, when the slotted segment is fetched, the slot's DP field is fixed to point to an access-protected reserved address range big enough to hold the entire object. The actual object data may be fetched from the network in one step, or dynamically as pages in the object's reserved address range are being accessed. A small table that associates slots of large objects with disk addresses is kept in the slotted segment.

Clearly, for "very" large objects the above scheme may not work; there may be memory size constraints that would make it impractical to build or access the whole object. Also, typically, very large objects are created in steps by successive appends while the scheme of the previous paragraph works only for objects whose size is fixed at creation time.

$\mathcal{B}$eSS offers a class interface for very large objects that includes byte range operations – such as `read`, `write`, `insert`, `delete` a number of bytes starting at some arbitrary byte position within the object, and `append` bytes at the end of the object. In anticipation of object growth, hints about the potential size of the object can be provided by the user. The large object is stored in a sequence of variable-size segments indexed by a tree structure [3, 4], and the root of the tree is placed in the overflow segment. Large objects created in such a way are not accessed transparently, i.e., as small objects. Instead, the user would have to use the interface provided in the class.

The advantages of the inter-object reference scheme and the segment and object structures employed by $\mathcal{B}$eSS are the following:

- Databases can be re-organized on the fly without affecting object references. Reorganization includes compaction, resizing, or relocation of data segments and movement of entire files between storage areas. This is an important issue because our system is planned to be used in a federated environment. In such an environment it is impossible to locate and change references to $\mathcal{B}$eSS objects from the other database management systems that participate in the federation. This is also useful in parallel architectures because it makes easy the I/O load balancing.

- Memory address space is reserved in a less greedy fashion than the schemes presented in [19, 30, 34]. In $\mathcal{B}$eSS, virtual address space for data segments is reserved only when the corresponding slotted segments are actually accessed.

- Persistent objects are manipulated directly on the segment on which they reside, without incurring any in-memory copying cost.

- Lock information is accessed via a pointer from the object header, rather that by a slower mechanism such as hash table lookup.

2.2 Preventing Database Corruption

In $\mathcal{B}$eSS, object references are virtual memory addresses. Thus, user code has direct access to $\mathcal{B}$eSS control structures such as slotted segments; and mechanisms to prevent database corruption caused by bad pointers are of paramount importance. Pointer errors are expected in database systems accessed by languages that have pointers, arrays or explicit dynamic allocation. These errors are especially troublesome because usually they cannot be detected immediately. Critical database structures can be corrupted by one transaction and discovered later in another completely unrelated one.

$\mathcal{B}$eSS utilizes the standard facilities provided by the underlying hardware for detecting access protection violations. The virtual memory management hardware detects an illegal attempt to update write-protected items at the time the update is attempted, before the possible error takes place and propagates to other structures. As shown in Figure 1, the slotted segment is mapped into write-protected virtual memory and thus ordinary user code cannot modify this memory section. Data segments are readable and potentially writable by user code. Before $\mathcal{B}$eSS (or some other trustworthy software) updates critical control structures, it explicitly unprotects the address space containing these structures, and it reprotects the address space after the update. This scheme allows correct software to modify protected data but prevents accidental (but not malicious) database updates by incorrect pointers[1]. The major cost associated with this kind of protection is an increased number of system calls [31], which for many applications is an acceptable tradeoff for the benefits gained.

2.3 Detecting Updates

$\mathcal{B}$eSS manages the locking of database pages in an automatic and transparent way by using the virtual memory protection mechanisms provided by the underlying hardware. When an application program gains access to a database page, $\mathcal{B}$eSS write-protects the virtual address space associated with this page. A protection violation is signalled by the hardware when the application attempts to modify that page and the $\mathcal{B}$eSS interrupt handler is invoked. The $\mathcal{B}$eSS interrupt handler records the update, performs locking, and grants write access to the page before the offending instruction is resumed. In this way, $\mathcal{B}$eSS automatically maintains the transactions' read and write sets and ensures that log records are written for all the modified pages.

In comparison, other storage systems (e.g., Exodus [28] and early implementations of EOS [8]) follow a software approach where the programmer explicitly indicates dirty data via a function call. This approach makes programming cumbersome and error prone – forgetting to invoke the function before an update may result in either incorrect results because of concurrent updates or loss of data because the updates were not propagated to disk. Also, when a compiler generates code for the storage system, the software approach is problematic due to separate compilations. For example, when an object pointer is passed as an argument to a C function, the C or C++ compiler has to make a conservative prediction that the object will be modified and thus, it generates an exclusive lock request – even though the object is not actually updated during the function execution.

Notice that hardware based detection works only for granules that are integral multiples of the page size used by the virtual memory system. We are currently examining issues related to object level locking [27]. Object level locking is realized by following a software-based approach.

2.4 Extensibility

One of our goals in designing $\mathcal{B}$eSS was for its architecture to be extensible. In a system with a modular architecture (e.g., such as [2], with "pluggable" components) new functionality can be added by replacing modules of the system. On the other hand, well designed object-oriented systems support modularity by separating interface and implementation.

$\mathcal{B}$eSS consists of a number of interacting modules with well-defined interfaces and has extensible facilities provided by the storage system itself. These facilities provide controlled access to a number of entry points in the system and allow applications to enhance and extent the functionality offered by $\mathcal{B}$eSS.

Consider a simple and pragmatic scenario. A user wants to count the number of transaction commits performed in a $\mathcal{B}$eSS system during some period of time. An impractical solution is to add a few lines of measurement code to the source of *every* application. Alternatively, the user could modify the $\mathcal{B}$eSS `commit` function to increment a counter each time a transaction successfully commits. But this solution requires user modification of the $\mathcal{B}$eSS internals.

$\mathcal{B}$eSS offers a better way. Programmers have controlled access to a number of entry points in the system via the notion of primitive events and hook functions. In this way, users may enhance or modify the behavior of $\mathcal{B}$eSS and their applications without changing the application code or changing the internals of the $\mathcal{B}$eSS system.

The hooks must be registered with $\mathcal{B}$eSS, usually before any access to persistent data is initiated. Examples of primitive events include segment fault or replacement, database open, locking, transaction commit, and deadlocks. In addition to the events that can be detected by its software components, $\mathcal{B}$eSS also traps the `SIGSEGV` and `SIGBUS` signals delivered by the underlying hardware when a virtual memory protection violation is caught.

A partial implementation of the above mechanism was part of early releases of EOS and has been used extensively. For example, the implementation of Ode [1] uses hooks to fix hidden pointers in persistent C++ objects[2]. Hooks have also been used to more effectively deal with very large objects by compressing them when they are stored on disk, and uncompressing them when they are fetched to memory. The compression/decompression functions are written by the user and registered with the $\mathcal{B}$eSS system.

2.5 Interface

Object retrieval is implicit – i.e., via dereference[3] – using a number of $\mathcal{B}$eSS typed references that are

[1] Clearly, if malicious software manages to unprotect memory before updating it, this protection mechanism alone would not work.

[2] These pointers are called hidden because they are not specified by the user [5]. They are placed by the compiler on objects of types that have virtual functions or virtual base classes, and they are invalid across program invocations.

[3] We use the term *dereference* for any indirect access – either through dereference operators such as `*` and `->` in C and C++, or through indexing an array or pointer variable.

based on the ODMG-93 standard [14]. For example, the C++ class `ref<T>` encapsulates a pointer to an object header as discussed in section 2.1. It is parameterized with the type of the pointer for which its instances can be substituted. Given a type, say, `Person` with members `name` and `spouse`, the instantiated type `ref<Person>` behaves like a pointer of type `Person`. So, a variable `p` of type `ref<Person>` can be used as if it were of type `Person*`; i.e., one can use `p->spouse->name` to refer to the `name` member of the `p->spouse` object, or pass `p` to a function argument expecting a `Person*` variable. Other kinds of type references are discussed in section 4.

An object may also be retrieved explicitly by supplying to 𝓑eSS the name of the object and a pointer to the database the object is stored. Any 𝓑eSS object can be given a name. For such so called "named" or "root" objects, 𝓑eSS maintains a directory which is implemented as a pair of hash tables. 𝓑eSS enforces the referential integrity between root objects and their names, e.g., when a root object is removed from a database so is the name of the object. Also, explicit retrieval can be performed using the class `global_ref<T>` that encapsulates an OID but access via this mechanism is somewhat slower compared to the one described in the previous paragraph.

Finally, new objects in 𝓑eSS are created by a number of overloaded functions. These functions require the size and a pointer to the type descriptor of the object being created, as well as where the object should be created – in a database, in a specific file, or in a specific object segment. They return a pointer to the object header of the newly created object, which may then be cast to the appropriate type[4].

3 The 𝓑eSS Distributed Architecture

𝓑eSS operates on a multi-client multi-server environment. A typical 𝓑eSS network configuration is illustrated in Figure 2. Application programs are executed either on client workstations (e.g. the applications running on node 1 and node 3) or on the same machine as a 𝓑eSS server (e.g. the application running on node 2). Application programs may need to interact with two kinds of 𝓑eSS processes during their execution: the *𝓑eSS server* and the *𝓑eSS node server*.

Each 𝓑eSS server manages a number of storage areas and it provides distributed transaction management, concurrency control and recovery for the databases stored in these areas. The *two phase commit* (2PC) protocol is employed for distributed commits and timeouts are used for distributed deadlock detection. The strict two phase locking algorithm is used for concurrency control and recovery is based on an ARIES-like [21] *write-ahead log* (WAL) protocol. Moreover, client-server interaction is minimized by caching data and locks between transactions running on the same client. Cache consistency is provided by employing the *callback locking* algorithm [17, 19],

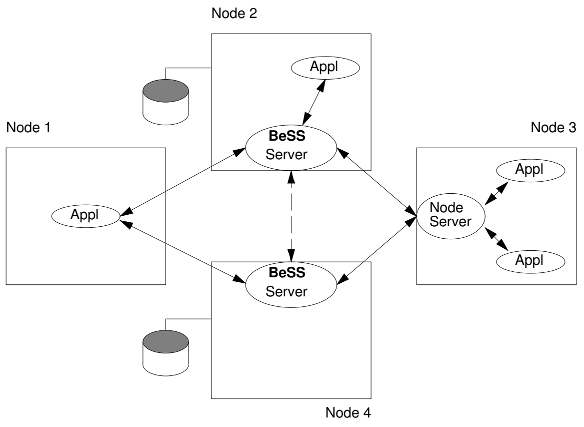

Figure 2: A network of 𝓑eSS servers and client workstations

which has been shown to have good performance over a wide range of workloads [32, 13].

A 𝓑eSS node server is a 𝓑eSS server that does not own any storage areas. Consequently, each 𝓑eSS node server is a client of the 𝓑eSS servers that acts as a server for the local applications. The 𝓑eSS node server establishes a cache on the node it is running and it is responsible for fetching the data requested by the local applications from the 𝓑eSS servers that own the data. In addition, the 𝓑eSS node server acquires locks on behalf of the local applications and responds to callback requests issued by 𝓑eSS servers.

Applications running on nodes with a 𝓑eSS server or a node server can access the entire distributed database space by communicating only with the local 𝓑eSS server or node server, respectively. Applications running on nodes without a 𝓑eSS server and 𝓑eSS node server may have to communicate with multiple 𝓑eSS servers in order to access a number of databases (e.g., the application running on node 1 in Figure 2). For those applications, data and locks are cached only during the duration of a transaction. When the transaction terminates, it releases all locks and cleans its private buffer pool. Furthermore, distributed transaction processing for each of those applications is performed by the first 𝓑eSS server the application establishes a connection with.

4 𝓑eSS Process Structure

Figure 3 illustrates the structure of the cache created in a node by the server or the node server. The cache is created by using the shared memory facilities provided by UNIX that associates a virtual address range with a file. The cache is viewed as a contiguous sequence of equal length frames, and the size of each frame is equal to the page size. Control data in the cache include lock tables, pending callback requests, transaction control structures, etc.

4.1 Operation Modes

A user process can access the shared cache either directly (*in-place access* or *shared memory*) or indirectly

[4]In languages such as C++ this can be done automatically by overloading the `new` operator. We do not further discuss this option.

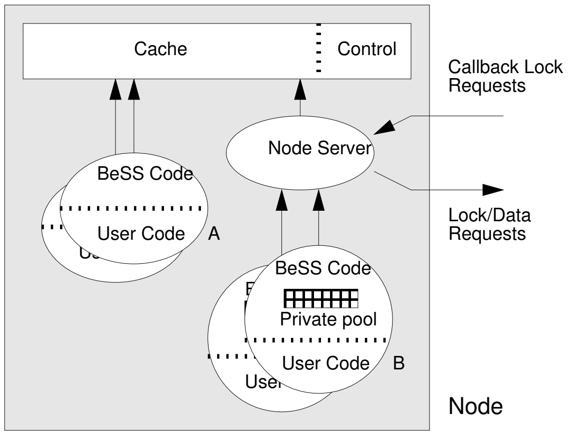

Figure 3: Shared memory established by the node server.

through the node server (*copy on access*). In the former case, each process gains access to the shared cache and all control data. In the latter case, each process maintains only a private cache (Figure 3, application B) and interprocess communication between the process and the node server is used to fetch segments in the private cache. If the segment is not present in the shared cache, the node server will retrieve it from the appropriate $\mathcal{B}$eSS server.

Copy on access has the advantage that user processes do not need to synchronize their accesses to their private caches, but inter-process communication is expensive. In-place access offers the potential for high performance, especially for short transactions, since it avoids interprocess communication and the cost of copying data to a private space and back to the cache. However, it incurs the cost of synchronizing concurrent access to the shared cache. The shared memory mode enables sophisticated users with well tested and debugged code to tailor the storage system and build multiple specialized servers, such as multimedia servers. Note also that the interface provided by the node server is the same in both modes, it is just the process boundaries that differ.

4.1.1 Copy On Access Mode

In the copy on access mode, each process has a private buffer pool to cache segments. The buffer pool is implemented as a fixed size file divided into a number of frames whose size is equal to the $\mathcal{B}$eSS page size. The above file is mapped into the process' virtual address space using the UNIX `mmap` system call. Because the file serves as backing store for the buffer pool, no physical or swap space is allocated for the virtual frames that are mapped into the file, although the size of the operating system's page tables increases. When the buffer pool becomes full, replacement takes place as described in Section 4.2.

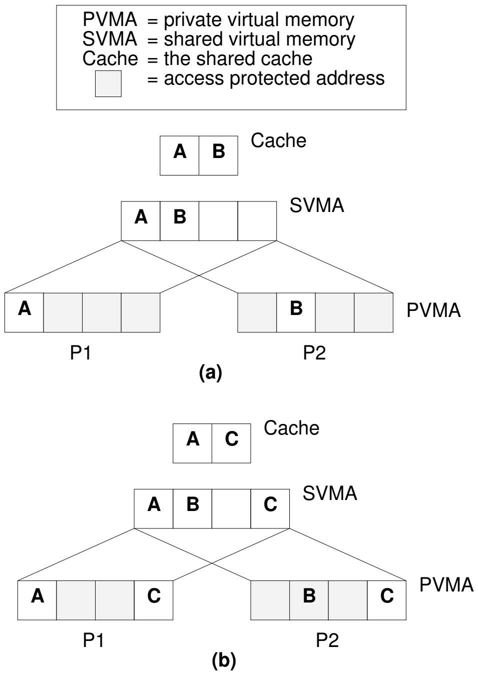

Figure 4: Implementation of shared virtual memory address space.

4.1.2 Shared Memory Mode

In the shared memory mode, apart from the problem of synchronizing concurrent accesses to the shared cache, pointers between database objects and their control structures, pointers among the control structures, and pointers among database objects must be valid to every application process accessing them. $\mathcal{B}$eSS uses latches (atomic test-and-set) for synchronizing concurrent accesses and implementing atomic read/write operations on the cached objects. Cleanup of shared structures from process failures is handled by keeping track of process actions as in [20].

$\mathcal{B}$eSS insures the validity of the shared pointers by treating them in a uniform way as offsets from the beginning of a fictitious virtual address space as outlined below and illustrated in Figure 4. Each process P maps the shared cache in a number of frames – each having size equal to database page – in the process' private virtual memory address range, referred to as PVMA. PVMA may be much larger than the size of the shared cache. Also, for our scheme to work all processes must reserve the same number of PVMA frames. Mapping of database pages to virtual frames is performed via a mapping table, referred to as SMT, *shared* by all processes. This means that if a process maps a page at some frame, all processes see this page at this frame (but possibly at different address).

Thus, pointers in the shared space are made valid by a) mapping each database page fetched in the shared cache to the same PVMA frame for all processes, and b) using offsets instead of virtual memory pointers. The shared mapping table in conjunction with the use of offsets gives the illusion of a shared virtual address space, referred to as SVMA. Note also, that in this scheme a pointer needs to be fixed once by the first process that fetched the corresponding page in cache. A simple $\mathcal{B}$eSS template class `shm_ref<T>` translates pointers from the process's virtual address space to pointers in the shared address space, and vice versa.

Figure 4 illustrates how two application processes $P1$ and $P2$ operate in the shared memory mode. Assume an empty cache accessed by two processes. When $P1$ wants to access page A the SMT assigns a virtual frame for this page, say the first one. Next, A is read into a cache slot and $P1$'s first PVMA frame is mapped to this cache slot. The same procedure is followed by $P2$ that wants to access page B and the outcome is depicted in Figure 4(a). Next, $P2$ wants to access page C. The SMT assigns an unused virtual frame to this page, say the last one, and B is replaced by the cache replacement algorithm, presented in Section 4.2, to make room for C. The replacement of B causes $P2$ to disable both read and write access to the PVMA frame that is mapped to B. When $P1$ wants to access C, the SVMA mapping indicates that the last PVMA frame should be mapped to the second cache slot that holds C. The result is presented in Figure 4(b).

4.2 Cache Replacement

$\mathcal{B}$eSS uses a clock-like (a.k.a. second chance) algorithm for page replacement policy [16]. However, $\mathcal{B}$eSS does not implement the traditional clock algorithm where a bit is kept for each slot in the cache, indicating whether or not the slot has been accessed since the last time the clock swept over it. This is because the cache manager does not have enough information indicating which slots have been accessed recently due to the memory mapping architecture.

$\mathcal{B}$eSS solution to the clock algorithm is based on the state of a virtual frame. Each virtual frame may be either *invalid,* or *protected,* or *accessible*. A frame is invalid when it is access-protected and it does not correspond to any cache slot. A frame is protected when it is access-protected and corresponds to a cache slot. Finally, a frame is accessible when it can be accessed by the application without causing an access violation. An accessible frame always corresponds to a cache slot.

The clock algorithm sweeps through the virtual frames and skips all invalid frames. Accessible frames are also skipped but after they are converted to protected. The cache slot corresponding to a protected frame is selected for replacement in the copy on access mode. In the shared memory mode, the above cache slot cannot be unilaterally replaced because it may being accessed by other processes.

Our solution to this problem is the following. $\mathcal{B}$eSS associates a counter with each cache slot. This counter corresponds to the number of processes that can access that slot and each process increments it when the process gains access to that slot. In addition, the clock algorithm is broken up in the following two levels.

The first level is the same as the clock algorithm in the copy on access mode with the difference that the protected frames are made invalid and the counter of the slot they correspond to is decremented by one. The second level operates on the cache slots and uses the counter as an indication whether or not the slot has been accessed since the last time the clock swept over it. A cache slot with counter zero is selected for replacement.

5 Related Work

In this section we draw comparisons with related work in the area of object-oriented and client-server systems.

Existing storage systems prevent database corruption by providing a function call interface to the client applications (e.g., Exodus [11]). On the other hand, $\mathcal{B}$eSS offers direct access to objects and utilizes the standard virtual memory facilities provided by hardware to detect access protection violations. $\mathcal{B}$eSS also uses the underlying hardware to automatically detect writes, as in ObjectStore [19, 24] and QuickStore [34].

Objects often contain references to other objects. Usually, the on-disk representation of an object differs from the in-memory representation of the object because of the way the storage manager treats inter-object references. Some systems use object identifiers (OIDs) for both the in-memory and on-disk representations [9]. Other systems use virtual memory pointers for both representations [29, 10, 19, 34]. Finally, there are systems that use OIDs for the on-disk representation and virtual memory pointers for the in-memory representation. These systems convert the OIDs to virtual memory pointers when they fetch the objects from disk – this is referred to as *swizzling* [23, 7]. The conversion can be done either in software [33, 22, 15, 26, 25] or by using the facilities provided by the underlying hardware [30, 35]. In $\mathcal{B}$eSS, inter-object references are represented by virtual memory pointers to the headers of the referenced objects which in turn contain the address of the object itself. Also, since $\mathcal{B}$eSS stores object headers in a different segment than the object data, segments containing object data can be moved around without affecting the inter-object references.

To meet the diverse needs of both present and future applications, it is very important for a storage system to be extensible. In advanced relational database management systems, extensibility usually stems from higher level system components such as query optimization. In terms of storage systems, GENESIS [2] extensibility comes in the form of building blocks that can be inserted into or removed from the system in order to meet application specific requirements. Extensibility in SHORE [12] is realized through the *value added server* concept. A trusted piece of code can be linked with the SHORE server to provide additional features to applications.

Similar to SHORE, the $\mathcal{B}$eSS server can be linked with code written by sophisticated users to provide specialized server facilities. In addition, $\mathcal{B}$eSS offers controlled access to a number of entry points in the system by allowing applications to specify the actions that should be taken when certain primitive events (software or hardware) occur.

Finally, $\mathcal{B}$eSS offers an adaptable process structure. The client-server system closest to ours is ObjectStore [19]. However, ObjectStore does not allow application processes to operate directly on objects that reside in the shared client cache. Instead, applications have to copy the items they need into their private space.

$\mathcal{B}$eSS differs in many respects from our previous work in the context of EOS [9, 8] although, it includes components of EOS that according to our experience have worked very well – disk space allocation, large objects, and user defined hooks are some of them. First, pointer dereference in EOS is somewhat slow because inter-object references are OIDs. $\mathcal{B}$eSS offers a fast pointer dereference mechanism by using virtual memory pointers. References within an object are located by looking up the object's type descriptor that is stored in the database, and they are swizzled to virtual memory addresses when the segment containing the object becomes accessible to the application. Second, object relocation in EOS is a tedious task because OIDs are physical addresses. On the other hand, $\mathcal{B}$eSS uses one level of indirection which facilitates on the fly database reorganization – compaction, resizing, and relocation of data segments – without affecting existing references. Third, unlike EOS that requires explicit calls for setting locks, $\mathcal{B}$eSS utilizes the virtual memory mechanisms provided by ordinary hardware for guarding against software errors, such as stray pointers, and for automatic lock acquisition. Fourth, $\mathcal{B}$eSS is an open-server system; i.e., user code can be linked with the $\mathcal{B}$eSS server to build application specific servers, as opposed to running the application as a client, the only alternative offered by EOS. Finally, $\mathcal{B}$eSS offers to application processes running on the same machine the capability of accessing data in a shared cache.

6 Conclusions

In this paper we have presented the architecture of the $\mathcal{B}$eSS storage system as well as a preliminary performance evaluation of the operation modes offered. The alpha implementation of $\mathcal{B}$eSS was completed in November 1993 and a beta release was completed in November 1994. $\mathcal{B}$eSS has been implemented in C++ and it is operational for SUN and SGI platforms. Also, we are planning on porting $\mathcal{B}$eSS on a multiprocessor machine such as the NCR 3600 with board level shared memory.

We are currently working on issues related to client-logging to offer high performance transaction management to client applications [27]. The $\mathcal{B}$eSS node server running on a node that has local disk space can exploit this space for logging purposes. In this way, the $\mathcal{B}$eSS node server will be able to commit local transactions, rollback local transactions, and recover from node crashes.

Acknowledgments

We would like to thank S. Dar, B. Hillyer, G. Linderman, and B. O'Connell for their valuable comment and helping us improve the presentation of this paper. Thanks are also due to N. Gehani and D. Lieuwen for their valuable comments.

References

[1] R. Agrawal and N. Gehani. ODE (Object Database and Environment): The language and the data model. In *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data,* Portland, Oregon, June 1989.

[2] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, SE-14(11):1711–1730, November 1988.

[3] A. Biliris. An efficient database storage structure for large dynamic objects. In *Proceedings of the Eighth International Conference on Data Engineering,* Tempe, Arizona, pages 301–308, February 1992.

[4] A. Biliris. The performance of three database storage structures for managing large objects. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data,* San Diego, California, pages 276–285, May 1992.

[5] A. Biliris, S. Dar, and N. Gehani. Making C++ objects persistent: The hidden pointers. *Software Practice and Experience*, 23(12):1285 – 1303, December 1993.

[6] A. Biliris, B. K. Hillyer, and E. Panagos. The Calico multimedia storage system. 1994. (Submitted for publication.).

[7] A. Biliris and J. Orenstein. Object storage management architectures. In A. Dogac, M. T. Ozsu, A. Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*, pages 185–200. Spring-Verlag, New York, 1994.

[8] A. Biliris and E. Panagos. EOS User's Guide, Release 2.0. Technical report, AT&T Bell Laboratories, May 1993.

[9] A. Biliris and E. Panagos. Transactions in the client-server EOS object store. In *Proceedings of the Ninth International Conference on Data Engineering,* Taipei, Taiwan, March 1995. To appear.

[10] P. A. Buhr, A. K. Goel, and A. Wai. μDatabase: A toolkit for constructing memory mapped databases. In *Proceeding of the Fifth Int'l Workshop on Persistent Object Systems,* San Miniato, Italy, pages 166–185, September 1992.

[11] M. Carey, D. DeWitt, G. Graefe, D. Haight, D. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. In S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–500. Morgan Kaufmann, San Mateo, California, 1990.

[12] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. McAuliffe, J. F. Naughton, D. T. Schuh, and M. H.

Solomon. Shoring up persistent applications. In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data,* Minneapolis, Minnesota, pages 383 – 394, May 1994.

[13] M. J. Carey, M. Franklin, M. Linvy, and E. Shekita. Data caching tradeoffs in client-server DBMS architectures. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data,* Denver, Colorado, pages 357–366, May 1991.

[14] R.G.G. Cattell. *Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, California, 1993. Contributions by T. Atwood, J. Dubl, G. Ferran, M. Loomis, and D. Wade.

[15] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):51–63, October 1991.

[16] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[17] J. H. Howard, M. Kazarand, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[18] F. Carino Jr., W. Sterling, and I. T. Leong. MoonBase - a complete multimedia database solution. In *Proc. of the ACM Multimedia '94 Conference, Workshop on Multimedia Database Management Systems*, pages 27 – 36, San Fransisco, California, October 1994.

[19] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):51–63, October 1991.

[20] D. Lomet, R. Anderson, Rengarajan T.K., and P. Spiro. How the Rdb/VMS data sharing system became fast. Technical Report CRL 92/4, Digital Equipment Corporation, Cambridge Research Lab, May 1992.

[21] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[22] J.E.B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[23] J.E.B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[24] Object Design Inc., Burlington, Massachusetts. *ObjectStore User Guide, Release 1.1*, March 1991.

[25] Objectivity Inc. *Objectivity/DB Documentation V2*, 1993.

[26] ONTOS Inc., Burlington, Massachusetts. *Ontos DB 2.2 Reference Manual*, 1992.

[27] E. Panagos, A. Biliris, H.V. Jagadish, and R. Rastogi. Exploiting client disks for high performance client-server architectures. 1994. Submitted for publication.

[28] J. E. Richardson. Compiled item faulting: A new technique for managing I/O in a persistent language. In *Proceeding of the Fourth Int'l Workshop on Persistent Object Systems,* Martha's Vineyard, Massachusetts, pages 3–16, September 1990.

[29] E. Shekita and M. Zwilling. Cricket: A mapped, persistent object store. In *Proceeding of the Fourth Int'l Workshop on Persistent Object Systems,* Martha's Vineyard, Massachusetts, pages 89–102, September 1990.

[30] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An efficient, portable persistent store. In *Proceeding of the Fifth Int'l Workshop on Persistent Object Systems,* San Miniato, Italy, pages 11–33, September 1992.

[31] M. Sullivan and M. Stonebraker. Using write protected structures to improve software fault tolerance in highly available database management systems. In *Proceedings of the Seventeenth International Conference on Very Large Databases,* Barcelona, Spain, pages 171–180, August 1991.

[32] Y. Wang and L. A. Rowe. Cache consistency and concurrency control in client/server DBMS architecture. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data,* Denver, Colorado, pages 367–376, May 1991.

[33] S. J. White and D. J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the Eighteenth International Conference on Very Large Databases,* Vancouver, British Columbia, pages 419 – 431, August 1992.

[34] S. J. White and D. J. DeWitt. QuickStore: A high performance mapped object store. In *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data,* Minneapolis, Minnesota, May 1994. To appear.

[35] P. R. Wilson and S. V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In *Int'l Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992.