follows the *page diffing* [10, 4, 13] approach, which presents an efficient way to locate modified portions on a page. Under the page diffing approach, the updated portions of a page are identified by comparing the updated copy of the page against a *clean* copy of the same page.

Log records are generated when the recovery buffer is full, or when an updated page is replaced from the client cache, or at transaction commit. The copy of the updated page in the recovery buffer is compared against the copy of the page in the cache for determining the regions that are different. Next, the before and after image of each modified region are used to generate the log record for the page. This log record is then inserted in an in-memory log buffer, which is flushed to the log at commit time or when it becomes full.

5 Conclusions

In this paper we have presented the architecture of the BeSS storage system. The alpha implementation of BeSS was completed in November 1993 and a beta release was completed in November 1994. BeSS has been implemented in C++ and it is operational for SUN and SGI platforms. Also, we are planning on porting BeSS on a multiprocessor machine such as the NCR 3600 with board level shared memory. Currently, BeSS is being used as the storage engine of the Teradata's content based multimedia system that provides an extended relational interface [9].

References

[1] 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.

[2] 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.

[3] 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.

[4] A. L. Hosking, E. W. Brown, and J. E. B. Moss. Update logging for persistent programming languages: A comparative performance evaluation. In *Proceedings of the Nineteenth International Conference on Very Large Databases,* Dublin, Ireland, pages 429–440, August 1993.

[5] 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.

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

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

[8] 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.

[9] W. O'Connell, T. Ieong, D. Schrader, C. Watson, G. Au, A. Biliris, S. Choo, P. Colin, G. Linderman, E. Panagos, J. Wang, and T. Walter. A Teradata content-based multimedia object manager for massively parallel architectures. In *Proceedings of ACM-SIGMOD 1996 International Conference on Management of Data,* Montreal, Canada, pages 68 – 78, May 1996.

[10] 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.

[11] 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.

[12] 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, pages 395–406, May 1994.

[13] S.J. White and D.J. DeWitt. Implementing crash recovery in QuickStore: A performance study. In *Proceedings of ACM-SIGMOD 1995 International Conference on Management of Data,* San Jose, California, pages 187–198, June 1995.

accessed since the last time the clock swept over it[2]. Instead, the clock algorithm is based on the state of a virtual frame. A virtual frame may be *invalid, protected,* or *accessible*. A frame is invalid when it is access-protected and it is not mapped to any cache slot. A frame is protected when it is access-protected and it is mapped to a cache slot. Finally, a frame is accessible when it is mapped to a cache slot and it can be accessed by the application. 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.

In the copy on access mode, a cache slot that corresponds to a protected frame is selected for replacement. In the shared memory mode, however, the cache slot of a protected frame cannot be unilaterally replaced because another application may be accessing it. For this reason, BeSS associates a counter with each cache slot. The counter corresponds to the number of applications that can access the slot. Each application increments the counter of a slot when it gains access to the slot.

In addition, the clock algorithm is broken up in two levels. The first level is the same as the clock algorithm in the copy on access mode with the difference that a protected frame is converted to invalid, and the counter of its slot is decremented by one. The second level operates on the cache slots and uses the counter as an indication whether the slot has been accessed since the last time the clock swept over it. A cache slot with counter zero is selected for replacement.

4 Transaction Management

BeSS provides two-phase locking for concurrency control with read/write lock for each page. Transactions involving more than one server are coordinated using the two-phase commit protocol, and timeouts are used for distributed deadlock detection. BeSS supports inter-transaction caching of data and locks. Cache consistency is maintained by using the *callback locking* algorithm [5, 6]. Recovery is based on the ARIES [8] write-ahead logging protocol.

4.1 Detecting Updates

BeSS detects updates by using the virtual memory management facilities provided by the underlying hardware. When a page is mapped in the application's virtual space for the first time, it is write-protected and a page descriptor is created for it. The page descriptor, which contains information about the mapped address of the page and the access permissions, is inserted into a balanced tree whose key is the virtual memory address of the page. An attempt to modify this page signals a protection violation and the BeSS fault handler is invoked.

The fault handler first makes sure that the address of the fault is within some virtual frame F handled by BeSS – if not, the fault propagates to the application. Next, based on the address of F, the above tree is searched to locate the descriptor of the page P corresponding to F. Then, a write lock is acquired on P, if needed, and the appropriate recovery actions take place. The following cases need to be considered:

1. F is access protected and P is not cached. P is fetched in the cache, as explained in Section 3.2, and the steps described in case 3 below are taken.

2. F is access protected because of the normal activity of the cache replacement mechanism, and P is cached. The mapping between P and F is re-established, F is write-protected, and control is returned to the application process.

3. F is write-protected and P is cached. The current image of P is copied into a separate space, referred to as *recovery buffer*, and the descriptor of P is made to point to this recovery buffer. Write access is enabled for F, and control is returned to the application process.

This approach offers a transparent, automatic, and fast way of detecting updates. The overhead associated with updates (locking and logging) occurs exactly once, when the page is updated for the first time. Any subsequent updates proceed at full speed. This is important for many applications, such as CAD, that typically work on many objects by repeatedly traversing relationships between these objects and updating some of them as well.

4.2 Generating Log Records

Although the virtual memory management hardware detects updates, the granularity of the detection is that of the page size supported by the virtual memory system – typically 4 Kilobytes. Consequently, the modified regions on a page cannot be detected. Generating a log record for the entire page, i.e. assuming the whole page is dirty, may have implications in the performance of the system [13]. For this reason, BeSS

[2] Because of the memory-mapped architecture, the cache manager does not have enough information to identify the slots that have been accessed recently.

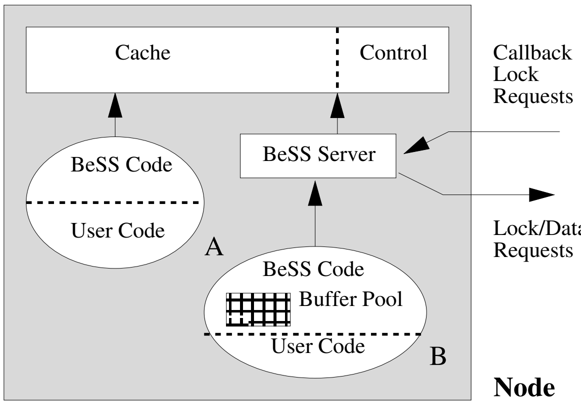

Figure 3: Modes of operation

plemented as a fixed size file that is mapped into the process' virtual address space. The file is divided into frames, each of which can hold a database page. Note that the interface provided by BeSS is the same for both modes; it is just the process boundaries that differ.

Copy on access has the advantage that user processes do not synchronize their accesses to their private caches. However, inter-process communication is required. Shared memory access offers the potential for high performance by avoiding inter-process communication and copying of data. However, it incurs the cost of synchronizing concurrent access to the shared cache. In addition, 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.

BeSS uses latches (atomic test-and-set) for synchronizing concurrent accesses to the shared cache. Clean-up of shared structures from process failures is handled by keeping track of process actions as in [7]. BeSS ensures the validity of the shared pointers by treating them as offsets from the beginning of a fictitious virtual address space. In particular, each process maps the shared cache in a number of frames – each having size equal to database page – in its private virtual memory address space (PVMA). The size of PVMA may be much larger than the size of the shared cache, however, for our scheme to work all processes reserve the same number of PVMA frames.

Each database page fetched in the shared cache is mapped to the same PVMA frame for all processes by using a mapping table that is *shared* by all processes. The shared mapping table together with the use of offsets give the illusion of a shared virtual address space.

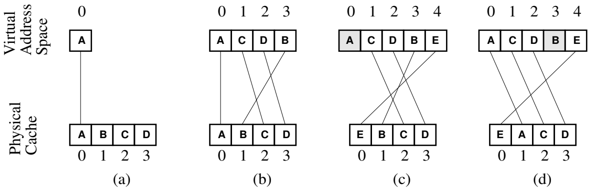

Figure 4: The BeSS memory mapping approach

Under this scheme, a pointer needs to be fixed once by the first process that fetched the corresponding page in the cache. A simple BeSS template class translates pointers from the process's virtual address space to pointers in the shared address space, and vice versa.

3.2 Memory Mapping

As we mentioned in the previous section, BeSS maintains a mapping from virtual frames to physical cache frames. Let us assume that a transaction needs to access page A that happens to be in cache frame 0, perhaps fetched by another transaction, as shown in Figure 4(a). Some virtual frame, say frame 0, is mapped to cache frame 0, pointers within the virtual frame 0 are fixed if needed, and the application gets read-only access to this frame.

The transaction keeps accessing pages in the fashion described above until the number of accessed pages reaches the capacity of the cache, as shown in Figure 4(b), where the transaction has mapped its virtual frames 0, 1, 2, and 3 to cache frames 0, 2, 3, and 1, respectively. Next, the transaction needs to access page E, which is not currently in the cache. As shown in Figure 4(c), page A is evicted from cache frame 0 and the virtual frame 0 is access protected; page E is fetched in cache frame 0, the virtual frame 4 is mapped to this cache frame, and the application gets read access to this virtual frame.

The cache-to-virtual frame mapping is dynamic in the sense that the same virtual address frame may be mapped to different physical frames during the execution of a transaction. For instance, if page A needs to be brought back in cache, it may be placed in any cache frame and the virtual frame 0 will be mapped to that new cache frame, see Figure 4(d).

3.3 Cache Replacement

Cache replacement in based on a clock-like algorithm. However, BeSS does not use the traditional clock algorithm where a bit indicates whether a slot has been databases can be re-organized[1] on the fly without affecting object references.

To illustrate how BeSS manages object references, consider the actions taken when a slotted segment is fetched in memory during the processing of a *slotted segment fault*. First, a virtual memory address range for its data segment is reserved and access-protected. Then, the DP field of every slot that corresponds to an object in the data segment – 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.

A *data segment fault* occurs when some object within a data segment, for which addresses have been reserved as discussed above, is accessed. Depending on the availability of cache space, the whole data segment or the part needed to access the object is fetched. Next, the type descriptor of every object contained in the fetched portion is examined, and each reference to some object O is updated to point to the virtual memory address of O's header. If the slotted segment containing O's header has never been referenced before in the current transaction, an address range is reserved and access-protected before the reference to O is modified. When O is accessed, a slotted segment fault occurs, and the actions described in the previous paragraph are repeated.

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 memory and address ranges for the corresponding data segments are reserved. Finally, accessing some objects within one of these data segments causes the data segment to be fetched. The latter may trigger another round of virtual memory address reservation and data fetching. Under this scheme, memory address space is reserved in a less greedy fashion than the schemes presented in [6, 10, 12].

2.4 Preventing Database Corruption

Since object references are virtual memory addresses, user code has direct access to BeSS control structures such as slotted segments. Hence, mechanisms to prevent database corruption caused by bad pointers are of paramount importance. BeSS uses the 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 2, slotted segments are mapped into write-protected virtual memory and ordinary user code cannot modify them. Data segments are readable and potentially writable by user code. Before BeSS (or some other trustworthy software) updates control structures, the address space containing these structures is explicitly unprotected before the update and reprotected after the update is over. This scheme allows correct software to modify protected data but prevents accidental database updates by incorrect pointers. However, if malicious software manages to unprotect memory before updating it, this protection mechanism alone would not work. The major cost associated with this mechanism is an increased number of system calls [11]. However, for many applications this cost is an acceptable tradeoff for the benefits gained.

3 Implementation Issues

In this section, we discuss implementation issues regarding the two operation modes, memory mapping, and cache replacement.

3.1 Operation Modes

Each BeSS server manages a cache (see Figure 3) which stores pages that have been accessed by local transactions. These pages may belong to databases managed by the local server, if any, as well as databases managed by remote servers. The cache is created using the `mmap` UNIX system call, and it is viewed as a contiguous sequence of cache frames, each of which can hold a database page. The server is responsible for forwarding requests made by local applications to remote servers (e.g., for fetching remote data), and for processing requests made by remote servers (e.g., callback requests).

A user process can access the shared cache either directly (*in-place access* or *shared memory*) or indirectly through the server (*copy on access*). In the former case, each process gains access to the shared cache and all control data by mapping the cache into its address space. In the latter case, each process maintains a private cache (Figure 3, application B) and communication with the local server is required for fetching segments. This private cache of each process is im-

[1] Reorganization includes compaction, resizing, or relocation of data segments and movement of entire files between storage areas.

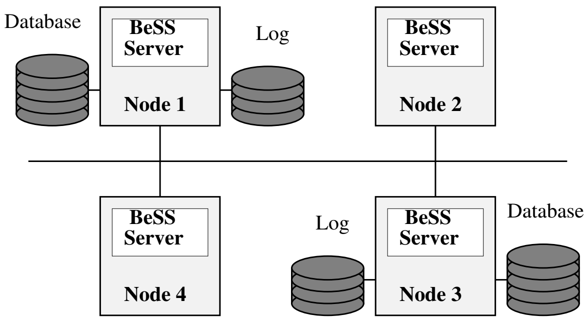

Figure 1: The BeSS distributed architecture

facility provided to users for physically co-locating objects on disk.

At the physical level, each database consists of one or more *storage areas*, which are either UNIX files or raw disk partitions. Storage areas are partitioned into several extents, and allocation of disk segments from one of these extents is based on the binary buddy system [1]. Storage areas that correspond to UNIX files may expand in size by one extent at a time.

At database creation time, the name of a storage area (i.e., the name of a UNIX file or a raw disk partition), which is referred to as the *root storage area*, must be provided. A database is identified by the name of the root storage area. All data structures needed for managing a database, such as space allocation maps, physical location of all storage areas belonging to the database, etc., are stored in the root storage area. Storage areas can be added to or removed from the database dynamically.

A BeSS file is stored entirely in single storage area, and each storage area may contain multiple BeSS files. To accommodate growth, BeSS files can be moved from one storage area to another, without affecting existing references to the objects within the moved files. In particular, a database may initially contain only the root storage area, and all user files are created in this area. As files grow, new storage areas can be attached to the database and some files can be moved to them. The maximum size of a single BeSS database is 128 Terabytes.

2.2 Segment and Object Structures

Each object segment consists of the *slotted segment* and the *data segment*, each of which is a sequence of one or more physically contiguous disk pages (Figure 2). Slotted segments are allocated from the root storage area and they are never relocated. Slotted segments contain a fixed-size header and an array of slots. Each header includes information required for managing the object segment – such as the number of objects, and the available free space in the data segment.

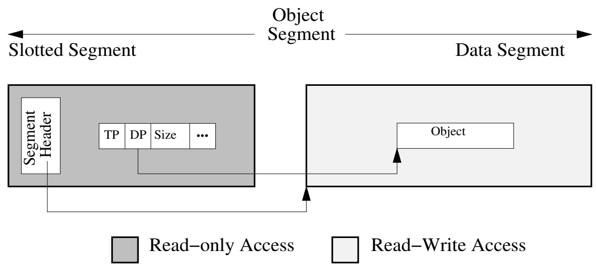

Figure 2: The structure of an object segment

Objects are stored in data segments. For each object there is an object header that is stored in a slot in the slotted segment. The object header contains 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), and the object's size. Type descriptors contain the offsets of all pointers stored in the objects they describe. Data segments can be re-sized or moved to a different location in the same or a different storage area. In the current implementation, the maximum size of a data segment is 64 Kilobytes.

Large objects, which cannot fit in a data segment, are stored in a sequence of variable-size segments indexed by a tree structure [1]. These objects are managed by using an interface 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. BeSS offers fast random access to an arbitrary byte within a large object [2], memory-type operations similar to UNIX `memcmp, memcpy, memchr`, asynchronous byte range operations, stripping over several storage areas, and prefetching.

2.3 Object References

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 that contains the complete address of the referenced object. The forward object is stored in the database of the referencing object. Inter-database references are handled transparently and exclusively by BeSS. The advantage of the above reference scheme is that

The BeSS Object Storage Manager: Architecture Overview

Alexandros Biliris and **Euthimios Panagos**
AT&T Research, Murray Hill, NJ 07974
{biliris, thimios}@research.att.com

Abstract

BeSS is a high performance, memory-mapped object storage manager offering distributed transaction management facilities and extensible support for persistence. In this paper, we present an overview of the peer-to-peer architecture of BeSS, and we discuss issues related to space management, inter-object references, database corruption, operation modes, cache replacement, and transaction management.

1 Introduction

BeSS, a **Be**ll Laboratories **S**torage **S**ystem, is a storage manager that facilitates the development of high-performance database management systems. The architecture of BeSS is not tailored to a specific data model or language. It is possible to build relational and object-oriented database systems as well as home-grown database systems and persistent languages on top of BeSS.

BeSS 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 [6, 10, 12]. However, as we shall see, our scheme does not involve a greedy allocation of virtual memory addresses. BeSS prevents database corruption caused by bad pointers by protecting control structures, which are stored separately from data, using the virtual memory management facilities provided by the underlying hardware.

An interesting aspect of the BeSS architecture is that it offers two operation modes for accessing persistent data: *copy on access* and *shared memory*. In the copy on access mode, applications operate on objects after copying them into a private buffer pool. In the shared memory mode, applications operate on objects present in a buffer pool that can be shared by many applications running on the same machine.

BeSS is based on a peer-to-peer architecture and provides distributed transaction facilities. BeSS offers concurrency control via locking and recovery via logging to support the traditional ACID transaction properties. Inter-transaction caching of both data and locks is supported, and cache consistency is guaranteed by using the callback locking algorithm [5, 6].

The remainder of the paper is organized as follows. Section 2 describes the architecture of BeSS, including storage structures, object references, and protection from stray pointers. Implementation issues are discussed in Section 3. Transaction management is covered in Section 4 and, finally, Section 5 concludes our presentation.

2 System Architecture

A typical BeSS network configuration is illustrated in Figure 1. Some nodes, such as nodes 1 and 3, own databases while some others, such as nodes 2 and 4, are client workstations that do not own any database. Similar to the peer-to-peer architecture of Shore [3], there is a BeSS server process running on every node. The presence of a BeSS server process on a node enables sharing of data across transactions that are part of either the same or different application processes running on this node. An application running on a node may access the entire distributed database space by communicating only with the local BeSS server.

2.1 Storage Entities

At the conceptual level, BeSS manipulates *databases*, which are collections of BeSS *files*. BeSS files are collections of *object segments* on which objects are stored. BeSS files group objects so that they can be retrieved one by one via a cursor mechanism. However, objects can be accessed directly, without accessing the file(s) containing them. Object segments are the clustering