A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems

by
David J. DeWitt
Philippe Futtersack
David Maier
Fernando Velez

Computer Sciences Technical Report #907
January 1990

A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems

David J. DeWitt
Philippe Futtersack
David Maier
Fernando Velez

Authors' affiliations: D. DeWitt, Computer Sciences Department, University of Wisconsin, currently on sabbatical with GIP Altair. D. Maier, Department of Computer Science and Engineering, Oregon Graduate Institute for Science and Technology, currently on sabbatical with GIP Altair. P. Futtersack and F. Velez, GIP Altair, France,

This work was partially supported by a research grant from The Graduate School of the University of Wisconsin (D. DeWitt) and by National Science Foundation grant IRI-8920642 (D. Maier).

ABSTRACT

In the engineering and scientific marketplaces, the workstation-server model of computing is emerging as the standard of the 1990s. Implementing an object-oriented database system in this environment immediately presents the design choice of how to partition database functionality between the server and workstation processes. To better understand the alternatives to this fundamental design decision, we analyze three different workstation-server architectures, evaluating them both qualitatively and through benchmarking of prototypes. The three approaches are labeled *object server*, in which individual objects pass between the server and workstation, *page server*, in which a disk page is the unit of transport and the server buffers pages, and *file server*, where whole pages are transferred as well, but they are accessed directly by the workstation process via a remote file service (namely, NFS).

We built prototypes of all three architectures, using a stripped-down version of the WiSS storage system as a starting point. To compare the performance of the prototypes, and to experiment with sensitivity to data placement and cache sizes, we developed our own object manager benchmark, the *Altair Complex-Object Benchmark* (ACOB). This benchmark supports experiments that vary both clustering (inter-object locality) and smearing (intra-object locality). The test suite of benchmarks includes queries for scanning the entire database and traversing and updating complex objects.

We present the results of several experiments run using ACOB on the three prototypes, along with results from a single-server version of WiSS that serves as a reference point. Our main conclusions are that the page-server and file-server architectures benefit most from clustering, that the relative performance of the page- and object-server architectures is very sensitive to the degree of database clustering and the size of the workstation's buffer pool relative to the size of the database, and that, while the file-server architecture dominates the page-server architecture on read-intensive operations, the opposite is true on write-intensive operations.

1. Introduction

While relational database systems were the technology of the 1980s, over past 1-2 years it has become apparent that object-oriented database systems are to become the key database technology of the 1990s. Harnessing the power of this technology will not be easy, as such systems pose many difficult engineering challenges. If one reflects on the commercialization of relational database systems, it took a full ten years to turn the first prototypes (Ingres and System R in 1976 [Ston76, Astr76]) into products that conservative customers willingly used. Given the relative complexity of object-oriented database systems, it is likely to take ten years before the technology of object-oriented database systems becomes solidified.

The situation is further complicated by the emergence of the workstation-server model of computing as the standard of the 1990s in the engineering and scientific market places — the initial target market for object-oriented database systems. While the early relational database system developers could safely ignore the issue of distribution, the developers of object-oriented database systems simply cannot. Thus, in addition to solving the many engineering challenges posed by object-oriented database systems, their developers must also tackle issues of distribution.

This paper examines one of the engineering challenges that must be solved: determining how the functionality of an object-oriented database system (OODBMS) should be distributed between the workstation and server. Why is this issue important? If we were implementing a relational system for such an environment, we would almost certainly implement a server process to which application programs running on the workstations would submit SQL queries. However, OODBMSs differ significantly from relational systems in the way that they manipulate data. With an OODBMS, a fair amount of an application is incorporated into the methods of the classes of which objects are instances. In addition, while OODBMSs also provide support for associative queries over sets of objects [Atki89], applications implemented using an OODBMS have a large navigational component. One might simply run all applications programs on a centralized server. Such an approach simply is not commercially viable as OODBMSs tend to be used for applications (e.g., CAD systems) that require the computational and graphical interface that only a dedicated workstation per user can provide. Thus, a centralized solution to the problem is not acceptable.

The remainder of this paper is organized as follows. Section 2 describes three alternative workstation-server architectures for an OODBMS and makes a qualitative comparison of the three. Section 3 presents our prototype implementations of these three architectures. The benchmark used to evaluate these alternative designs is given in Section 4 and the results of applying this benchmark to the three prototype systems are presented in Section 5. Section 6 contains our conclusions and recommendations.

2. Three Alternative Workstation/Server Architectures

2.1. Introduction

There appear to be at least three markedly different approaches for architecting an OODBMS in a workstation-server environment, and, within each general approach, a wide number of variations. One alternative is the **object server** approach, so named because the unit of transfer between the server and the workstation is an object. In this architecture the server understands the concept of an object and is capable of applying methods to objects. The V1.0 prototype of O_2 [Banc88, Deux90], the Orion 1 prototype [Kim90], and some pre-release versions of GemStone [Cope84] employ an object-server architecture.

An alternative is the **page server** approach. In this design, the server deals only with pages and does not understand the semantics of objects. Thus, it cannot apply methods to objects. In addition to providing the storage and retrieval of database pages, the server also provides concurrency control and recovery services for the database software running on the workstations. This architecture is currently being used by the ObServer [Horn87] and Exodus [Care89] prototypes.

The third design represents a further simplification of the page-server architecture in which the workstations use a remote file service, such as NFS [Sun88], to read and write database pages directly. As with the page-server design, the server in this architecture provides concurrency control and recovery services. The current version of GemStone uses this architecture when configured for a workstation-server environment. We term this architecture the **file server** approach.

In the following subsections we compare these three alternative architectures qualitatively, from the viewpoint of how each affects the implementation of the functionality that an OODBMS must provide. It will be obvious that there are many issues left unanswered. We have decided to concentrate our efforts on first deciding which of the three basic architectures provides the best overall performance and have ignored certain equally important issues, such as what is the best recovery strategy for any particular architecture. We will study such issues in the future.

2.2. The Object-Server Architecture

In the object-server architecture, most of the OODBMS functionality is replicated on the workstation and the server as shown in Figure 1[1]. In addition, both maintain caches of recently accessed objects, and methods can be applied to objects on either the workstation or server. When the workstation needs an object it first searches its local

[1] This is one of many possible variations of this architecture.

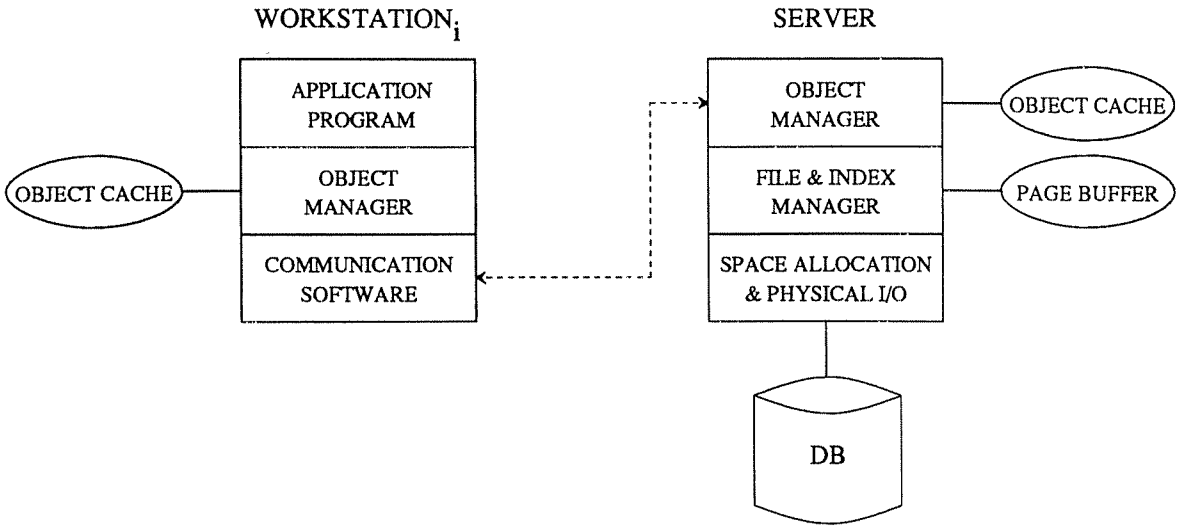


Figure 1

object cache for the object. If the object is not found it sends a request for the object to the server. If the object is not in the server's cache, the server retrieves the object from disk and returns it to the requesting workstation. The unit of transfer between the workstation and the server is an individual object. The server may or may not keep a copy of the desired object in its local cache, depending on the cache policy that it implements.

This architecture has a number of advantages. First, both the server and workstation are able to run methods. A method that selects a small subset of a large collection can execute on the server, avoiding the movement of the entire collection to the workstation. This ability is most valuable when there is no index on the collection that can be used by the selection operation. In addition, one can balance the system workload by moving work from the workstation to the server [Vele89, Bern89]. Another advantage is that this architecture simplifies the design of the concurrency control subsystem as the server knows exactly which objects are accessed by each application. Hence, concurrency control can be almost completely centralized in the server. Furthermore, the implementation of object-level locking is straightforward. The design might also lower the cost of enforcing constraint that relate objects manipulated by an application with other objects in the database, for example, a constraint that says that the subpart graph of a mechanical assembly is acyclic. Instead of moving all the objects involved in the constraint to the workstation, the constraints may be checked at the server.

This design suffers from some serious problems. First, in the worse case there may be one remote procedure call (RPC) per object reference, although hopefully the hit rate on the workstation's object cache will be high enough to satisfy most requests. For the server to transfer more than a single object at a time, it must be capable of figuring out which objects belong with one another, replicating the work that any clustering mechanism did when it placed objects together on a disk page.

Another major problem is that this architecture complicates the design of the server. Instead of having the server supply just the functionality that it alone can supply (e.g., sharing, concurrency control, recovery), the server must be capable of executing arbitrary user methods. In the case of the V1.0 O_2 prototype, this meant the use of System V shared memory (read: expensive and slow) for the server's object cache and lock table.

A related problem arises when a method is applied to a group of objects that are currently distributed among the workstation's cache, the server's cache, and the disk drives which hold the database. This problem is complicated by the possibility that the same object may be in both caches as well as on the disk simultaneously. In addition, an updated version of the object may exist in the workstation's cache but not in the server's cache. Thus, a method cannot blithely execute on the server, without first addressing such cache inconsistencies. O_2 V1.0 and Orion adopted different solutions to this problem. In the case of O_2, when a method (such as an associative query) is to be run on the server, the workstation flushes all the modified objects in its object cache back to the server (a fairly slow process). The Orion-1 prototype instead executes the method on both the workstation and server [Kim90]. Since this strategy can result in duplicate copies of an object in the result, a complicated (and probably expensive) postprocessing step of duplicate elimination is required.

There are also several minor problems that occur with the object-server architecture. First, it may hinder locking at the page level as an alternative to object-level locking since neither the workstation nor server software really deals with pages. Second, objects tend to get copied multiple times. For example, an object may have to be copied from the server's page-level buffer pool into its object cache before it can be sent to a workstation. Another copy may take place on the workstation if the object cannot be stored directly into the object cache by the networking software. Finally, since the software on a workstation simply requests an object from the server without generally being aware of the size of the object, large, multipage objects may move in their entirety, even if the application needed to access only a few bytes.

Finally, this design could be out of step with technology trends. Within 1-2 years, 10 MIP workstations will become increasingly common and most (>90%) of the computing power will end up being concentrated in the workstations and not in the server. In such a situation it makes no sense to move work from the workstation to the server. Rather, the functionality of the server should be stripped to its bare minimum. Likewise, over the next 5-10 years, FDDI will probably replace Ethernet as the standard technology for local area networks. While the cost of sending a message is already almost entirely due to software overhead and not transmission time, the overhead component will definitely dominate in the future.

2.3. Page-Server Architecture

Figure 2 shows one possible architecture for a page server. In this architecture, the server basically consists of a large buffer pool, the I/O level of a storage system for managing pages and files, plus concurrency control and recovery servers. The upper levels of the OODBMS software run exclusively on the workstation and the unit of transfer between the workstation and server is a disk page. When the server receives a request for a data page, it first sets the appropriate lock on the page. Next, if the desired page is not in the server's buffer pool, the server retrieves the page from the disk and returns it to the workstation.

Buffering on the workstation can be done either in terms of pages or in terms of objects or both. The advantage of an object cache is that buffer space on the workstation is not wasted holding objects that have not actually been referenced by the application. If an object cache is used, then the cost of copying each object from the incoming data page into the local object cache will be incurred. Furthermore, if an object in the object cache is later updated, the page on which the object resides may have to be retrieved from the server again (which, in turn, may incur an I/O operation).

Like the object server, this design has its advantages and disadvantages. Its primary advantage is that it places most of the complexity of the OODBMS in the workstation, where the majority of the available CPU cycles are concentrated, leaving the server to perform the tasks that it alone can perform — concurrency control and recovery. Since entire pages are transferred intact between the workstation and the server, the overhead on the server is minimized. While on first glance this approach may appear wasteful if only a single object on the page is needed, in fact the cost (in terms of CPU cycles) to send 4K bytes is not much higher than the cost of sending 100

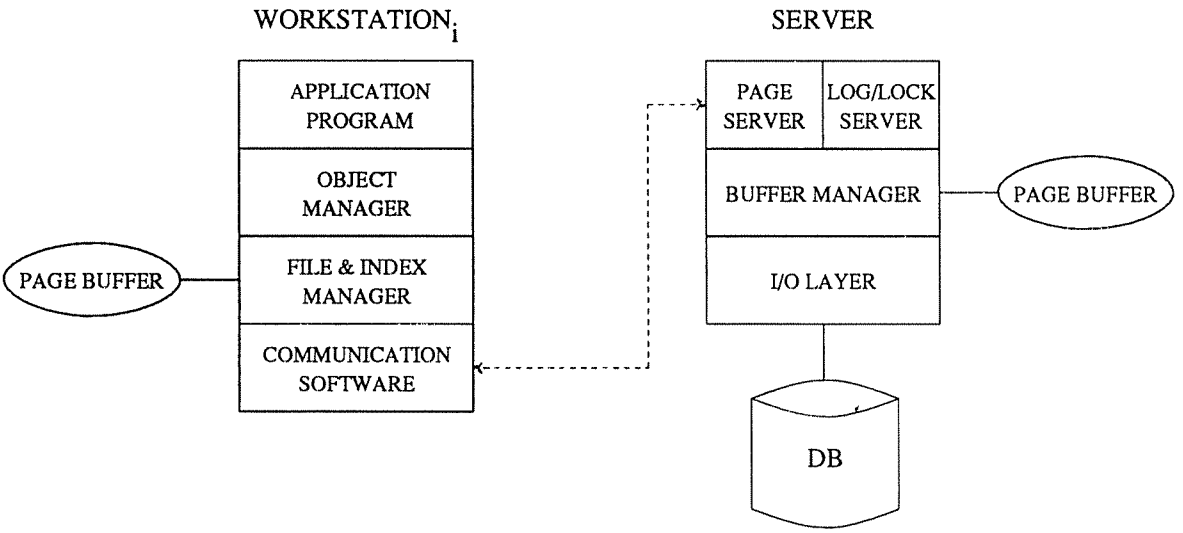


Figure 2

bytes. Furthermore, if the clustering mechanism has worked properly, then a significant fraction of the objects on each page will eventually end up being referenced by the client. Finally, by minimizing the load each individual workstation places on the server, one can support more workstations off a single server — delaying for a long as possible the complexities associated with supporting multiple servers.

While its simplicity makes this design very attractive, it is not without disadvantages. First, methods can be evaluated only on the workstation. Thus, a sequential scan of a collection requires that all the pages in the collection be transferred to the workstation. While this limitation sounds disastrous, there are several mitigating factors. First, the server does the same amount of disk I/O as with the object server design and, in the case of an indexed selection, only those pages containing relevant objects will be transferred to the workstation. Furthermore, the page server avoids all problems (e.g, having to flush the workstation's cache) that the object-server architecture encounters when executing methods on the server.

A second disadvantage is that object-level locking may be difficult to implement. Consider what happens if object-level locking is employed and two workstations, A and B, update two different objects on the same data page. If classical locking protocols [Gray76] are used, both transactions will set IX locks on the shared page and X locks on the individual objects. If special precautions are not taken to merge the changes to the two pages when the modified pages are written back to the server, one or the other of the updates will be lost. One possible solution, is to allow only one object-level X lock on a page at a time (by making IX locks on a page incompatible). In addition, implementing non-2PL B-tree locking protocols may be complex in this environment.

Finally, the performance of this design, both relative to the other architectures and in absolute terms, may be dependent on the effectiveness of the clustering mechanism. In Section 5, we examine how clustering effects the performance of this design.

2.4. The File-Server Architecture

The final workstation-server architecture is a variation of the page-server design in which the workstation software uses a remote file service such as NFS [Sun88] to read and write database pages directly. Figure 3 shows such an architecture.

There are several reasons why such an architecture is attractive. First, it provides many of the advantages of the page-server architecture, such as minimizing the overhead placed on the workstation by the server. Also, by using NFS to read and write the database, user-level context switches can be avoided completely, improving the rate at which data can be retrieved by a remote workstation. Finally, because of its widespread use, NFS will continue to evolve and be improved. Basing a system on NFS takes advantage of these improvements.

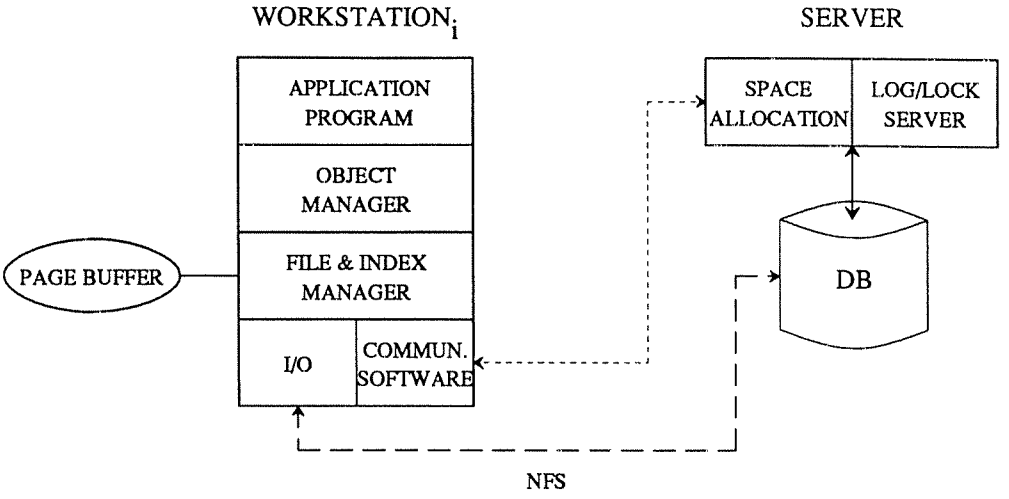


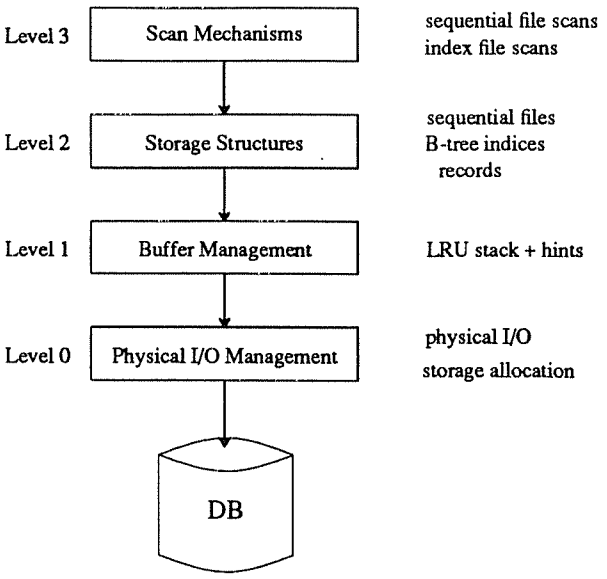
Figure 3

In addition to the problems it inherits from the page-server architecture, this architecture has some other serious problems. First, NFS writes are known to be slow. Because it is a stateless protocol built on top of UDP, a write operation to a remotely mounted NFS file is flushed to disk before the request is acknowledged. Second, since read operations in this architecture bypass the server software completely, it is not possible to combine the request for a page with the request for a lock on a page. While one can send a separate lock request message, such an approach negates some of the benefits of using NFS in the first place, as the cost of the lock request is likely to be very close to the cost of simply requesting the page and setting the lock as a side effect. Less costly alternatives are to use an optimistic concurrency control scheme or to batch lock requests to reduce the overhead of setting locks. A similar problem occurs with coordinating the allocation of disk space on the server. Since sending a request to the server for an individual page is too expensive, the most reasonable solution is to have the server allocate groups of pages at a time, instead of workstations requesting a single page at a time.

3. Prototyping The Workstation-Server Architectures

3.1. Introduction

This section describes our prototypes of the three alternative architectures. The basis for each of these prototypes was a stripped-down, single-user version of WiSS [Chou85]. We elected to use WiSS because it is currently being used as part of two different OODBMSs (the O_2 system from Altair and the ObjectStore system from Object Sciences). As shown in Figure 4, WiSS consists of four distinct levels. The lowest level, Level 0, deals with the aspects of physical I/O, including allocation of disk extents (collections of physically contiguous pages) to files. The next level, Level 1, is the buffer manager which uses the read and write operations of Level 0 to provide buf-



WiSS Architecture
Figure 4

fered I/O to the higher levels of the system. Pages in the buffer pool are managed using an LRU replacement strategy. Level 2 is the storage-structure level. This level implements sequential files, B^+-trees, and long data items. This level is also responsible for mapping references to records to the appropriate references to pages buffered by Level 1. Finally, Level 3 implements the access methods, which provide the primitives for scanning a file via a sequential, index, or long-data item scans.

To simplify the implementation of the different prototypes, we first produced a stripped-down version of WiSS by removing support for long data items, B-trees, and all but the most primitive scans. The resulting system supports only records, files, and sequential scans without predicates. In addition to serving as the basis from which the various prototype systems were constructed, this version of WiSS also provides a lower bound against which the performance of the other systems can be compared. The "ideal" workstation-server version of this system would have exactly the same performance as the single-processor version.

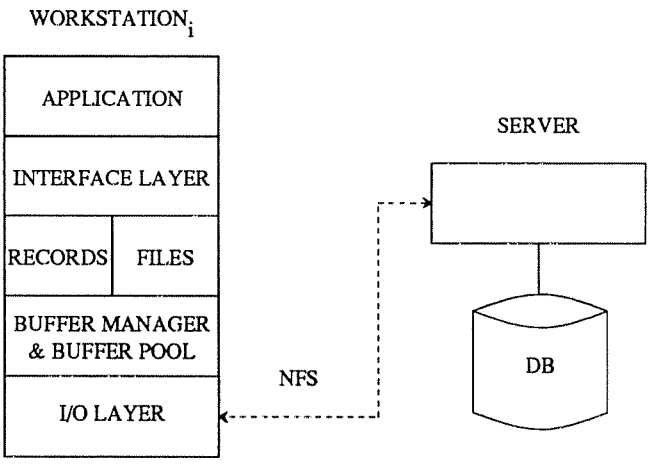
As discussed in more detail below, the object and page servers required that we split the functionality provided by WiSS into two processes, termed the **client** and **server** processes. In the target environment, the database resides on disks attached to the machine running the server process and the client process runs on each workstation. To provide the interprocessor communication necessary to implement such a system, we used the Sun RPC tools, including Rpcgen [Sun88], which automatically generates the necessary procedure stubs given a data file that describes the message formats and the procedure names with which you wish to communicate remotely. While this RPC package provides support for communications using both TCP and UDP, we elected to use UDP since we had

no need to send streams of data larger than 8 Kbytes.

3.2. File Server

As described in Section 2, the simplest way of having a number of workstations share data is to run all the database software on the workstation and use NFS to access the database on a shared server. Since for these experiments we elected to ignore issues of concurrency control, recovery, and coordinating the allocation of disk space, prototyping this architecture simply meant running WiSS on one processor with its disk volume on a remotely mounted file system belonging to the server, as shown in Figure 5. When Level 0 of WiSS makes an I/O request to read or write a page, the NFS software takes care of executing the request on the server on behalf of the workstation.

A key motivation for prototyping this design is that it provides a lower bound on how fast remote data can be accessed. While NFS, like the version of the Sun RPC protocol that we used, is also implemented on top of UDP, data transfers occur between two kernels instead of between two user-level processes. As illustrated by the results in Section 5, this difference can have a significant effect on the execution of certain kinds of queries.



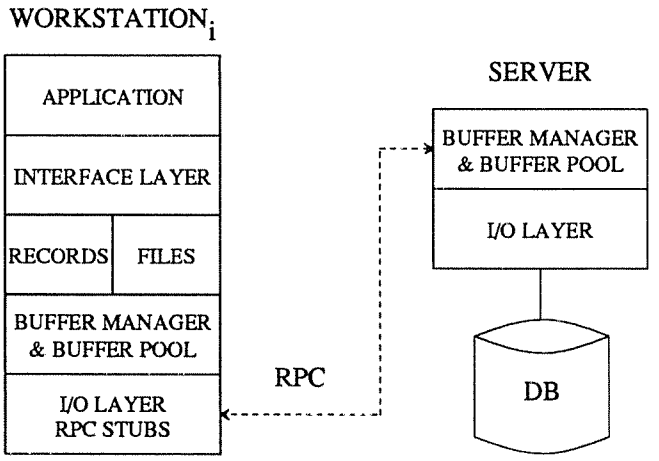
Remote File Server Design
Figure 5

3.3. Page Server

We prototyped the page server architecture as shown in Figure 6. The server process consists of Levels 0 and 1 of WiSS plus an RPC interface to all the routines in these two levels. The workstation process consists of all the levels of WiSS except for Level 0, which was replaced by an thin layer that executes requests to Level 0 by sending a remote procedure call to the server process. The upper levels of WiSS were completely unaffected by these changes. While our current implementation of this design does not provided concurrency control and recovery ser-

vices, the server does support simultaneous workstation processes.

In a complete implementation of this design, each request for a page would also specify the lock mode desired by the workstation (e.g., S, X, IX, ...), avoiding the overhead of a second message to set a lock. In addition to a centralized lock table on the server, each workstation would also maintain a local cache of the locks it holds to minimize lock calls to the server.



Page-Server Design
Figure 6

The success of the page-server design is predicated on two factors: the extent to which "related" objects are clustered together on a page and the relative cost of fetching an individual object versus a page of objects. As illustrated by Table 1, most of the cost of obtaining a page from the server is in the overhead of the RPC and communications protocols and not in transmission and copying costs.[2]

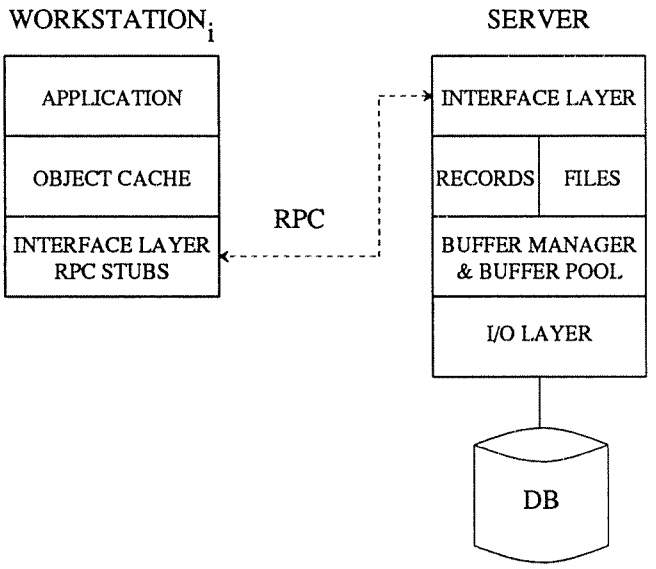
Size of Reply Message	Execution Time
1 byte	7.8 ms.
10 bytes	7.9 ms.
100 bytes	8.2 ms.
1000 bytes	9.7 ms.
4000 bytes	17.9 ms.
8000 bytes	28.4 ms.

Table 1

[2] These times were gathered between two Sun 3/80 processors running the Sun RPC software and Version 4.0.3 of the SunOS. The size of the RPC request message was 8 bytes.

3.4. Object Server

While the server process in the page-server architecture only understands pages, in the object-server design, it understands objects and files or collections of objects. Our object-server prototype is depicted in Figure 7. The server process consists of all the layers of WiSS plus an RPC interface to Level 3 of WiSS. The workstation process consists of the application code, a special version of Level 3 of WiSS, and an object cache. This version of Level 3 serves as an interface to the local object cache and to the services provided by the server process through the RPC software. Operations on files (e.g., create/destroy, open/close) are passed directly to the server. Operations on objects (really WiSS records) are handled as follows. In the case of a "read object" call, the Interface Layer first checks whether the object is in the local object cache. If so, the object is returned directly to the application code. When a miss on the local cache occurs, the Interface Layer first makes space available in the local cache (perhaps by writing an dirty object back to the server) and then requests the desired object from the server. When a new object is created, a "write-through" cache protocol is used because WiSS uses physical object-ids and the server is needed to place the object on a page of the appropriate file and assign an object-id. The new object remains in the cache but it is marked as clean. Updates to objects residing in the cache are handled simply by marking the object dirty and flushing any remaining dirty objects to the server at commit time.



Object_Server Design
Figure 7

3.5. Discussion

While our prototypes do not provide concurrency control or recovery services, since lock requests can be combined with object and page requests by the object- and page-server designs, respectively, the implementations

are sufficiently complete to allow a fairly thorough evaluation. Since lock requests cannot be combined with I/O requests in the file-server design, the results obtained for this design are undoubtably biased in favor of it (unless an optimistic concurrency control suffices).

4. The Altair Complex-Object Benchmark

4.1. Introduction

To evaluate the performance of the alternative workstation-server architectures, the obvious alternatives were to use an existing benchmark or to design a benchmark specific for this evaluation. In terms of existing benchmarks, the Sun [Catt88] and Hypermodel [Ande90] benchmarks appeared to be the most reasonable alternatives. We opted against the Sun benchmark because it forms complex objects by choosing random objects to relate to one another. Since one measurement we wanted was the impact of the degree of clustering of the components of a complex object on the performance of the alternative designs, the Sun benchmark was not appropriate.

Initially, the Hypermodel benchmark appeared a better match to our objectives because it provides both clustered and non-clustered groupings of objects. The problem with this benchmark is that it consists of a very large number of queries. Since we were afraid of being overwhelmed by results if used the full Hypermodel benchmark and did not understand which subset to select, we elected to design a new benchmark that was tailored to the task of evaluating the alternative workstation-server designs. In designing this benchmark, we borrowed a number of ideas from the Hypermodel and Sun benchmarks, but attempted to limit the scope of the benchmark to a simpler database design and a smaller set of queries.

We do not consider our benchmark an OODBMS benchmark, but rather a distributed object-manager benchmark. We make the distinction because the database generator has a number parameters controlling the physical placement of objects, such as the degree of clustering and the placement of records on pages, that are not expressible at the data model and data language level of an OODBMS.

4.2. Database Design

The basis for the Altair Complex-Object Benchmark (ACOB) is a set of 1500 complex objects. Each object is composed of 7 WiSS records with the structure shown in Figure 8.[3] Each record is 112 bytes long. The key field is an integer whose value ranges between 1 and 1500, corresponding to the physical order of the objects in the file.

[3] The design of this benchmark is influenced by the data model of the O_2 database system [Deux90], which distinguishes between values and objects. In an O_2 database, an object may have a complex structure without forcing each sub-component to be an object. Thus, it made sense for us to distinguish inter-object and intra-object references and treat them differently. In particular, we justify putting the 7 records of an object on one page as we imagine them all to be part of the value of one object, and hence would be created together when the object is created.

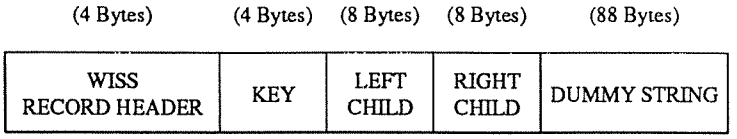
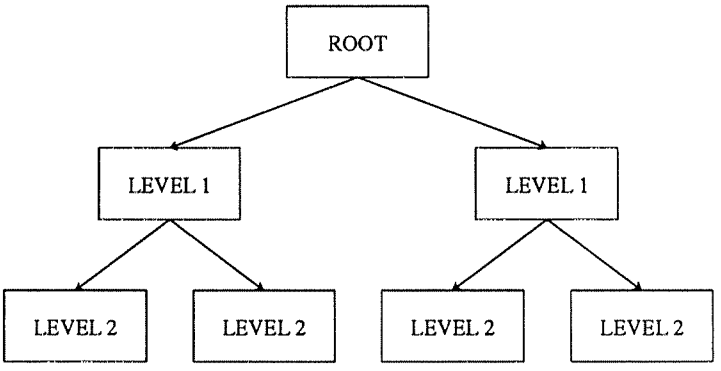


Figure 8

The seven records that form each complex object are organized in the form of binary tree of depth 2, as shown in Figure 9. All inter-record (and inter-object) references are in terms of physical record IDs. When an object is initially created, all 7 records are placed on the same data page. Since an update to a complex object may have the unavoidable side-effect of moving one or more of its component records to a different page, our benchmark provides a mechanism by which a fraction of the records within an object can be "smeared" to other pages in order to explore the effect of such updates on the performance of the different server architectures. Smearing is discussed in more detail later.

With 4 Kbyte pages, 5 complex objects (consisting of 7 records each) will fit on a single page and the set of 1500 objects spans 300 pages (approximately 1.2 megabytes). In order to minimize the effect of operating system buffering of database pages, 5 identical sets of 1500 objects were used for the experiments described below, producing a total database size of about 6 megabytes.

The ACOB database is constructed in three phases. The first phase allocates all the records of all the objects, forming the intra-object references and filling in all the record fields. In real object-oriented databases, objects frequently reference one another in either the form of aggregation relationships (e.g., a part and its component subparts) or M:N relationships between each other (e.g., suppliers and parts) [Ande90]. To simulate such relationships,



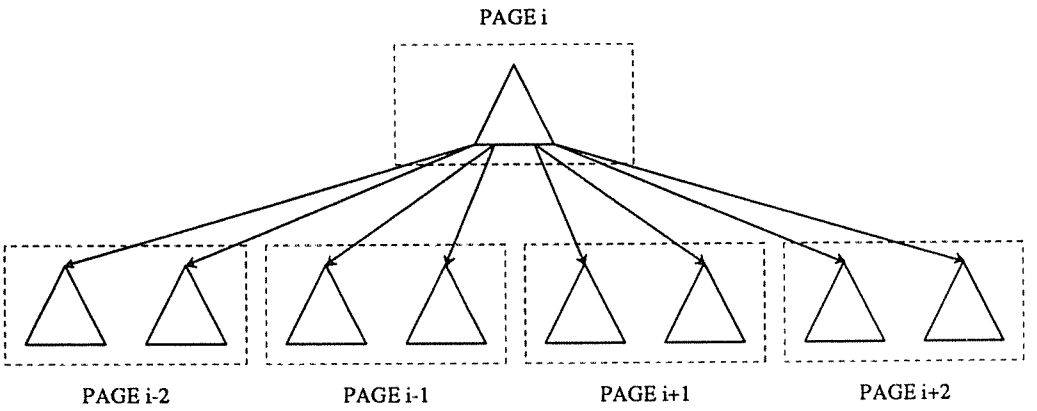
Complex Object Organization
Figure 9

in the second phase we *attach* two objects to each of the four leaf records in every complex object (via the "left" and "right" fields found in each record), as shown in Figure 10. (Each triangle in Figure 10 represents one complex of 7 records, as in Figure 9.) These attached objects are termed "components".

As mentioned before, we want to gauge the impact of physical clustering of complex objects on the performance of the different server designs. To study this effect, the second phase accepts a parameter called the **clustering factor**. Our notion of clustering is defined as follows. If object A references object B, then we say object B is **clustered** near object A if B is located within a "clustering region" around A. For example, if the size of the clustering region is 5 pages, B is considered to be clustered near A if B is on the same page as A or on either of the two pages physically preceding or following the page upon which A resides. Section 5 explores the effect of varying the size of the clustering region on the relative performance of the different architectures.

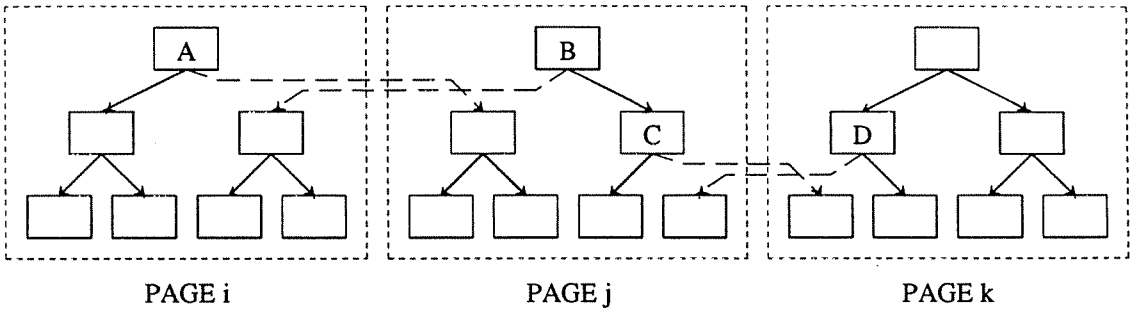
The clustering factor, f, can be varied from 0 to 100 and is employed as follows during the second phase of database creation. When the objects to attach to an object X are being selected, objects that are "clustered" nearby X are selected f% of the time. That is, for each attached object a random value between 0 and 99 is generated and if this random value is less than the value of the clustering factor, one of the objects that are clustered near X is selected at random. Otherwise, a random object from the set of 1500 objects is selected.

The third (optional) phase in creating the database is to "smear" a fraction of the records that comprise each complex object. Smearing simulates the dispersal of records in an object over different data pages caused by database updates (either because an updated record was too long to still to fit on its current page or because an alternative version of a record was created and the new version does not fit on the same page with the other records of the



A Complex Object with Its 8 Component Objects
(Clustered in a 5-Page Region)
Figure 10

object). If smearing is selected, one quarter of the objects (chosen at random) are modified as follows. With a probability of 0.25^4, each of the two internal records and each of the eight leaf records are "moved" to another page in the database. As shown in Figure 11, this "move" is accomplished by swapping the record with a corresponding record of another randomly selected object. For example, in Figure 11, we have smeared object A by exchanging its right child with the left child of B. In a similar fashion, we have smeared a leaf node from object B by exchanging the right child of record C with the left child of record D.



"Smearing" of Interior and Leaf Records within Several Complex Objects
Figure 11

4.3. Queries

Sequential Scan

The "scan" query reads all the complex objects in a set in their physical order. For each object, a breadth-first traversal algorithm is used to read the seven records that comprise the object[5]. This query does not, however, read any of the attached components of an object. Thus, each record is accessed exactly once. As records are accessed, they are copied into the address space of the application program. If the records of an object have been "smeared", reading an object may require access to more than one disk page.

This query has been included because it simulates reading all instances of a class. Without smearing, the query is a very good test of the instruction path length of each design. Without smearing, accesses to the database are strictly sequential, maximizing the chance that the performance of the CPU will be the limiting factor. This query, with its sequential I/O activity, was also designed to help understand the relative costs of accessing data on a

[4] The probability of being moved is computed individually on each node. However, since smearing involves swapping an internal record of one object with an internal record of another object, approximately 3/8ths of the objects are actually affected by the smearing process.

[5] In our implementation of this benchmark, the record IDs of the root records are stored in a separate set. In the case of the sequential scan query, the execution times presented in Section 5 include the time necessary to read the elements of this set. For the Random Read and Random Update queries, this set of root record IDs was read into an in-memory array as a separate processing step. The processing time for this step is not included in the execution times presented for these queries.

disk directly attached to the processor (i.e., WiSS running on a local disk) and accessing it remotely (i.e., each of the server designs). The degree of database clustering does not have any effect on the execution time of this query.

Random Read

The second query "processes" 300 randomly selected complex objects and their components. The following sequence of operations is applied to each object selected. First, the root record of the object is read. Then, a partial, depth-first traversal is performed up to a tree depth of 6 (the root record is considered to be at depth 0) such that an average of 44 records from among those in the object, its 8 attached components and the 64 root records of each of their attached components are read. This partial traversal is performed by electing to perform each stage of the depth-first traversal with a probability of 0.8. Thus, the expected number of records read at each level is 1 root + 0.8 of the children at depth 1, 0.8^2 of the records at depth 2, ... , and 0.8^6 of the records at depth 6. Of the expected 44 records reads, 17 are at level 6. Our selectivity factor on this query (and also the next) is related to the *structural density* parameter of Chang and Katz [Chan89]. Our selectivity of 0.8 appears to be in the middle of the range of structural densities actually observed in traces of access patterns from a suite of VLSI tools.

This query was designed to simulate those applications in which a user "checks out" a collection of complex objects and selected subcomponents at the beginning of a transaction or session. As we will see in Section 5, the extent to which the database is clustered has a significant effect on the execution time of this query for both the page- and file-server designs.

Random Update

The third query is basically an update version of the Random Read query. Again, 300 random objects are selected for processing but, in this query, the records at depth 6 that are selected for reading are also updated. On the average, for each complex object processed, 17 records are updated (out of the 44 records read). These updates are performed in place by overwriting the record with a copy of itself. Thus, the structure of the database is not changed.

This query was designed to help understand some of the possible benefits of transferring individual objects between the workstation and the server. With the page- and file-server designs, complete pages, containing perhaps only a single updated record, must be transferred back to the server from the workstation. In the case of the object server, only the updated objects need to be transferred back to the server. On the other hand, since the updated object must be placed back on the proper data page, but this page may no longer be in the server's page cache. In that case, the page must first be reread from the disk before the updated object can be added to it. One thing we wanted to explore was whether the benefits of transferring a single object were offset by increased I/O activity on

the server. This query is affected by the degree to which the database has been declustered and the extent to which the records in each object have been smeared.

4.4. Benchmark Organization

Using the load program plus the three queries described in the previous section, we then composed the actual benchmark that was used to evaluate the different server designs. This benchmark consisted of the following five steps that were always executed in the order described below. The random number seed was set to the same value at the start of each step. As described in Section 5, a wide number of versions of this benchmark were constructed by varying such parameters as the degree of clustering, whether smearing was selected or not, and the size of the workstation and server buffer pools.

Step 1 Build five identical sets of 1500 complex objects each with the same degree of database clustering and the same choice (either on or off) of smearing. The five sets are assigned names A to E corresponding to the order in which they were constructed (thus A is the first set built and E is the last set built).

Step 2 The second step in the benchmark is to apply the scan query on sets A through E in order. The reason for this ordering is to attempt to minimize the extent to which buffering of pages by the operating system affects the results obtained. We elected to use five sets of objects based on the relative sizes of the set of objects and the operating system buffer pool. The Sun 3/80 used as a server for our experiments had only 8 megabytes of memory. With a server with more memory we would have either increased the number of objects in each set or expanded the number of sets employed.

Step 3 The third step is to run the random read query on sets A through E, in order.

Step 4 The fourth step is to run the random update query on sets A through E, in order.

Step 5 The final component of the benchmark is to run the sequential scan, random read, and random update queries one after another on the same set (without reseting the random number generator between queries); first on Set A, then on the other sets in order. One motivation for this final phase of the benchmark was to explore how effectively the different server designs utilized their local buffer caches.

5. Performance Evaluation

In this section we present the results of our performance evaluation of the three prototypes using a single-site version of WiSS as a reference point. In Section 5.1, we describe the hardware environment used to conduct our experiments. Results obtained while building the ACOB database are contained in Section 5.2. Section 5.3 describes our clustering and smearing experiments. Finally, in Section 5.4, we explore how the size of the buffer pools on the workstation affects the architectures individually and comparatively.

5.1. Test Environment

For our experiments we used two Sun 3/80 workstations, each running Version 4.0.3 of the Sun operating system. Each machine had 8 megabytes of memory and two 100 megabyte disk drives (Quantum ProDrive 105S). While the machines were run in the normal, multiuser mode, during the tests they were not used for any other activity. The database was always constructed on the same disk drive to insure that any differences among the dif-

ferent drives did not affect the results obtained.

In configuring the different systems, a difficulty arose in deciding how much buffer space to allocate to WiSS and the file-server prototype.[6] Our default configuration for the page- and object-server designs was a 50-page (4 Kbyte pages) buffer pool on both the workstation and the server. One option would have been to allocate 100-page buffer pools to WiSS and the file-server design. However, this choice did not seem quite fair, because in a real environment the buffer space on the server would be shared among multiple workstations. In addition, 50-page buffer pools on both the workstation and the server are not as effective a single 100-page buffer pool. Since there was no obvious "right" solution, we used the same-size buffer pool as was used for the workstation with the page and object servers. This fact should be kept in mind when interpreting the results presented below. In addition, keep in mind that if standard 2PL locking were added to the file-server design (by having a lock server) its performance would degrade, perhaps significantly.

5.2. Database Build Time

In Table 2, the time for each of the four prototypes to build an unsmeared database with 30% and 90% clustering factors is presented. Several observations are in order. First, each system is affected only slightly by the clustering factor. While both the file- and page-server designs are slower than WiSS, the results are not out of line, considering that both are building the database on a disk attached to a remote processor. The most startling result is that the object-server prototype is almost a factor of 6 slower than the file- and page-server prototypes. As will become evident in the other tests presented below, when the object server's cache is ineffective (as with the build test), its performance is always much worse than the other designs because of the high cost of transferring objects individually. We omitted the times to build a smeared database (which are somewhat higher), because the relative performance of the different systems remained the same.

System	30% Clustering Factor	90% Clustering Factor
WiSS	22.0 secs.	22.6 secs.
File Server	47.8 secs.	48.4 secs.
Page Server	43.6 secs.	41.4 secs.
Object Server	236.8 secs.	233.8 secs.

Table 2

[6] One problem we encountered while developing the file-server prototype was limiting the amount of buffering performed by the operating system as Version 4.0 of SunOS uses all of available memory to buffer NFS pages locally. Consequently, the file-server prototype had an effective buffer-pool of 1,500 pages. Since there is no clean way of turning this feature off, we modified the kernel used for the file-server prototype by "setting" the size of the physical memory of the workstation machine to 292 pages. (This was done by using adb to set the value of _physmem to 0x175.) This value was chosen so that there was enough physical memory to hold the unix kernel, the benchmark program, a 50 page WiSS buffer pool, a 10 page NFS buffer pool, and the standard Unix utility programs.

5.3. Clustering and Smearing Tests

For the next collection of tests, we fixed the workstation and server buffer pools each at 50 pages and varied the clustering factor from 10% to 90%. The WiSS and file-server buffer pools were also set at 50 pages. The clustering region was fixed at 5 pages. Figures 12 through 19 present the results that we obtained for the four benchmark queries (see Section 4.3) on both unsmeared and smeared databases.

Scan Query

Consider first the results obtained for the scan query (Figures 12 and 13). As expected, each design is unaffected by the clustering factor since this query reads each object in its entirety, but none of an object's component objects. The most startling result is the object server's extremely poor performance relative to the other designs. With an unsmeared database, the page server is approximately 11 times faster than the object server. The source of this difference is the high cost associated with fetching each of the 7 records composing an object with a separate RPC operation. With the page-server design, 300 pages are fetched from the server by the workstation, one RPC call per page. With the object-server design, a total of 10,500 RPC calls are made.

When the database is smeared, the performance of the object-server design is relatively unaffected while the page and file-server prototypes slow down significantly. Part of this increase comes from a five-fold increase in the number of disk I/O operations required (to approximately 1500). However, this factor accounts for only 6 seconds[7] of the approximately 18 and 30 second increases observed for the file- and page-server architectures, respectively. The remainder represents the cost of fetching an additional 1200 pages from the server. For the page-server architecture, the cost of retrieving 1200 pages via the RPC mechanism can be estimated from Table 1 to be approximately 22 seconds. The 12 second difference between the two illustrates how much more efficient NFS is than a user-level RPC mechanism.

The object server is unaffected because it fetches the same number of records whether the database is smeared or not and the increased I/O activity on the part of the server is masked by the cost of doing object-at-a-time fetches.

---

[7] This is the increase observed for WiSS. The careful reader will notice that it is impossible to perform 1500 disk I/Os in 6 seconds, an average of 4 ms. per I/O. As discussed in the previous footnote, when we initially ran these experiments we were not aware of the fact that SunOS 4.0 uses all of memory to buffer disk pages. Thus, the server for each of the architectures actually had the use of approximately 6 megabytes of memory to buffer disk pages. When this fact was discovered we debated whether we should rerun all the experiments with a restricted server memory size. While doing so would have provided slightly more realistic results, we elected against making this change for two reasons. First, each design had an equal opportunity to use this memory for buffering. More importantly, restricting the amount memory on the server would have made the benchmark much more I/O intensive and this change might have masked the interesting differences among the architectures that we were already observing. The reader should keep this fact in mind when interpreting the results presented throughout this paper.

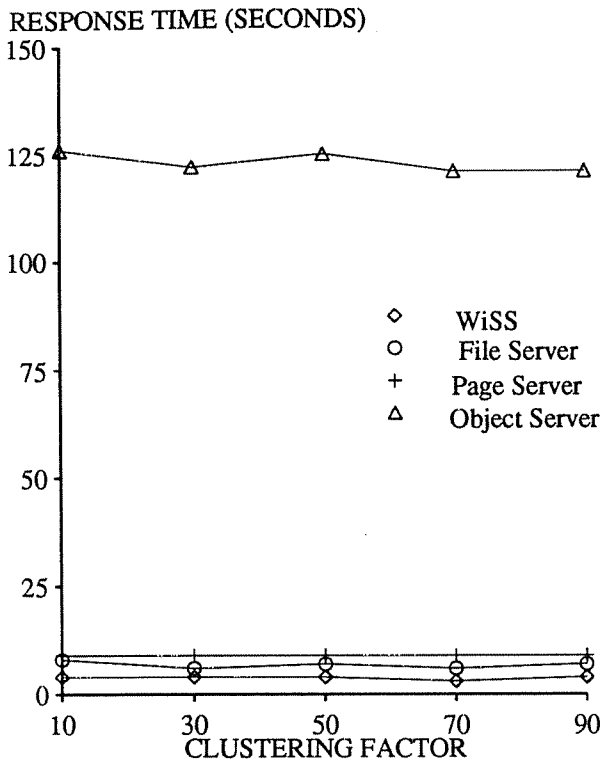


Figure 12: Scan Query, Unsmeared DB

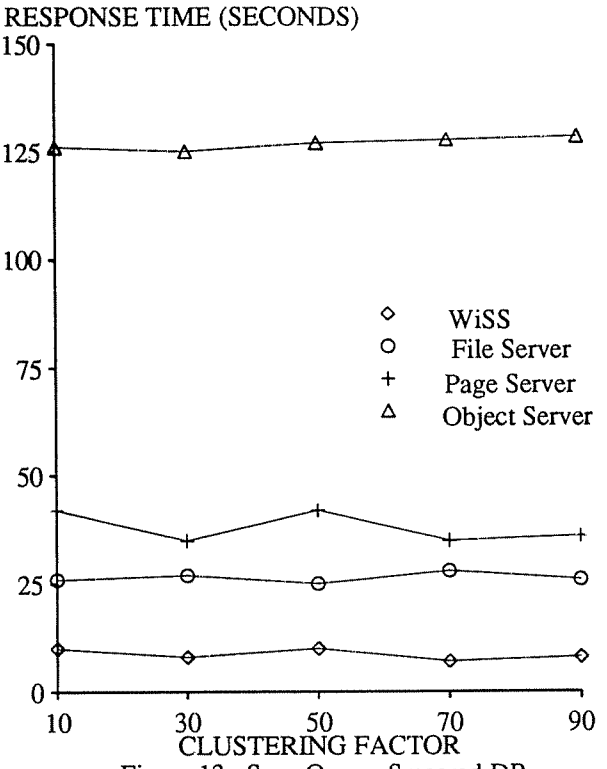


Figure 13: Scan Query, Smeared DB

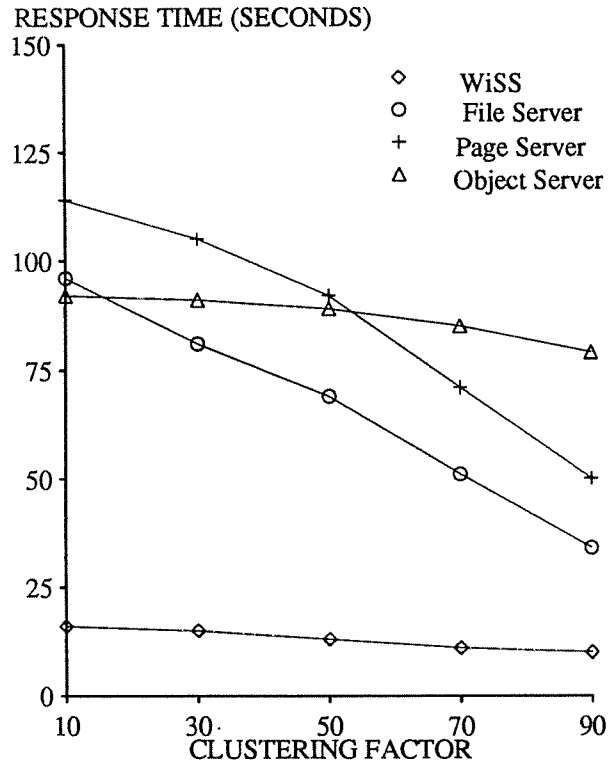


Figure 14: Random Read Query, Unsmeared DB

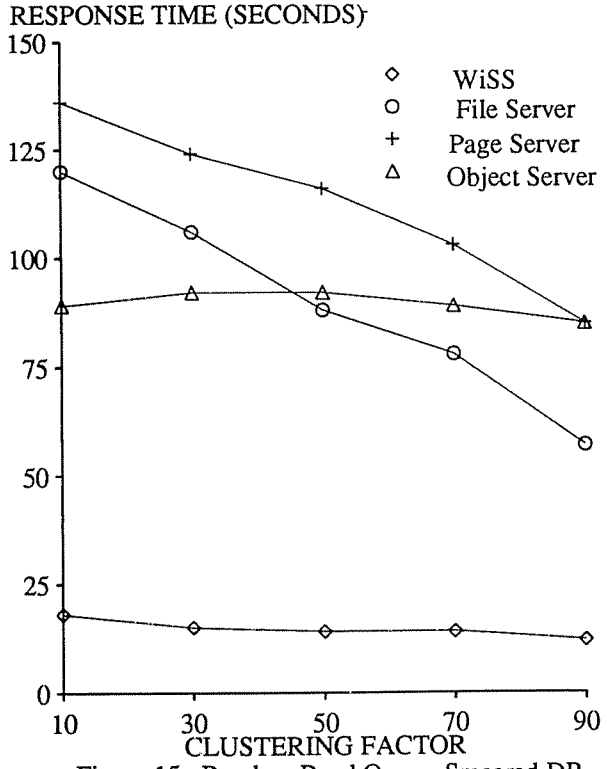


Figure 15: Random Read Query, Smeared DB

Random Read Query

Next consider the performance of the random read query for both smeared and unsmeared databases (Figures 14 and Figure 15, respectively). As the clustering factor is increased from 10% to 90%, the number of disk I/Os performed by each design to execute the query decreases (the buffer pool hit rate improves by approximately 30%). This reduction improves the performance of WiSS by about 35%.

The behavior of the object- and page-server designs (both in absolute terms and relative to one another) is more complicated. First, while both systems have 50-page buffer pools in both the workstation and the server (increasing the probability that an object will be found in one of the two buffer pools), it turns out that buffer size is not the dominant factor. Instead their behavior is dominated by the costs associated with RPC access to the server. The execution of this query involves 14,623 references to WiSS records (this is independent of the clustering factor). With a 10% clustering factor, the object server observes a cache hit rate of 0.181. When the clustering factor is increased to 90%, the cache hit rate increases to 0.251. The result is that 1024 fewer objects are requested from the server (10,953 instead of 11,977), producing an estimated RPC savings of only 8 seconds (the total improvement in response time is 13 seconds). The remaining 5 seconds is due primarily to the reduction in the number of disk I/Os performed by the server.

With the page-server design, as the clustering factor is increased from 10% to 90%, the buffer cache hit rate increases from 0.66 to 0.86, resulting in a savings of 3150 RPCs (from 5443 to 2293). This savings translates into an estimated RPC savings of approximately 58 seconds (consider the RPC times in Table 1, Section 3.3). The remaining 6 seconds improvement in response time is because the server does fewer I/Os.

The differences in performance between the page- and file-server designs is primarily due to the difference in cost between fetching a page via NFS and with an RPC call.[8] When the clustering factor is low (or the database is smeared as in Figure 15), the page-server design is forced to do a large number of RPC accesses, each of which is more expensive than accessing a page via NFS.

In general, the performance of the page- and file-server designs are very sensitive to the database clustering factor. The page-server design has worse performance than the object-server design with either a smeared database or a low clustering factor, simply because each page retrieved from the server ends up fetching relatively few objects that get referenced before the page is removed from the buffer pool. On the other hand, the object server fetches only those objects it actually references. Hence, its performance is relatively immune to the clustering fac-

[8] This difference is not the result of extra copies, as we modified the output of rpcgen to eliminate the the copy normally introduced by the rpcgen software.

tor or smearing. In a multiuser environment, this immunity would likely change if the bottleneck switches from being the RPC mechanism to the disk arm (i.e., if each object fetched results in a random disk access).

Random Update Query

Like the random read query, this query randomly selects 300 objects for processing, but, in this case, the records at depth 6 that are selected for reading during the traversal are also updated. On the average, for each complex object processed, 17 records are updated. The results are contained in Figures 16 and 17 for unsmeared and smeared databases, respectively.

This query illustrates the extremely poor performance of writes using NFS — the file server being more than a factor of 8 slower than WiSS at a clustering factor of 10%. As the clustering factor is increased from 10 to 90%, the response time for the file-server design improves to being only a factor of 5 worse than WiSS for an unsmeared database and a factor of 7 worse for a smeared database. This improvement occurs because with a higher clustering factor the chance that a page ends up being written more than once decreases. It is important to keep in mind that these results reflect the performance of a file-server design implemented using NFS. A different remote file service might behave quite differently.

The clustering factor and smearing again significantly affect the performance of the page-server design and, since pages flow both directions between the workstation and the server, the impact is magnified. Furthermore, the same page may end up being written back to the server more than once.

This query is the only one where the object-server design really exhibits significantly better performance than the other two designs. By fetching only those objects that are actually referenced by the application and writing back only the objects actually modified, the performance of the object-server design is relatively insensitive to clustering and smearing.

This query also illustrates that page writes with NFS are more expensive than page writes with RPC (at a clustering factor of 10%, the response time for the file server is about 20% higher than that of the page server). This difference is due to NFS being a stateless protocol: it performs the write operation before acknowledging the message. With an unsmeared database, the difference in response times decreases slightly as the clustering level is increased, because fewer pages are written more than once. (Recall that, at each clustering factor level, the same number of pages are being written by both architectures.)

All Queries Tests

The final test involves running the scan, random read, and random update queries sequentially without flushing the workstation or server's cache between the queries. The purpose of this query is to simulate the case where a

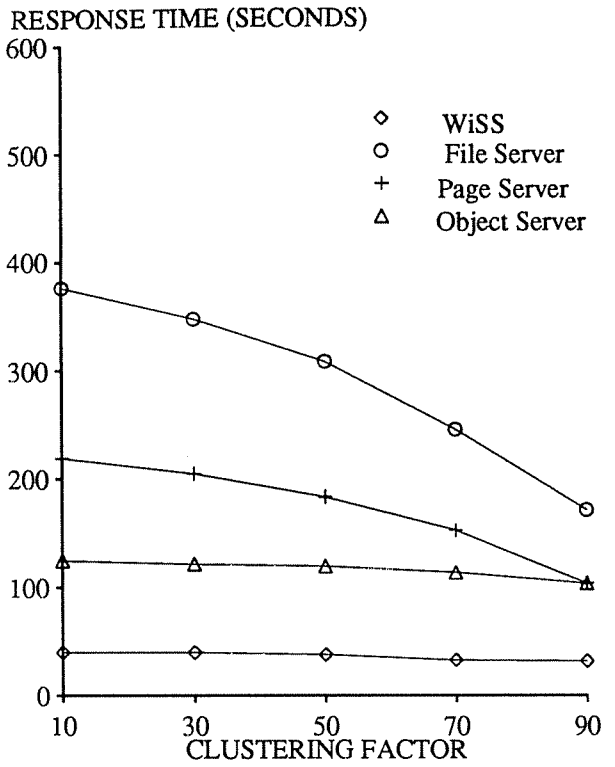


Figure 16: Random Update Query, Unsmeared DB

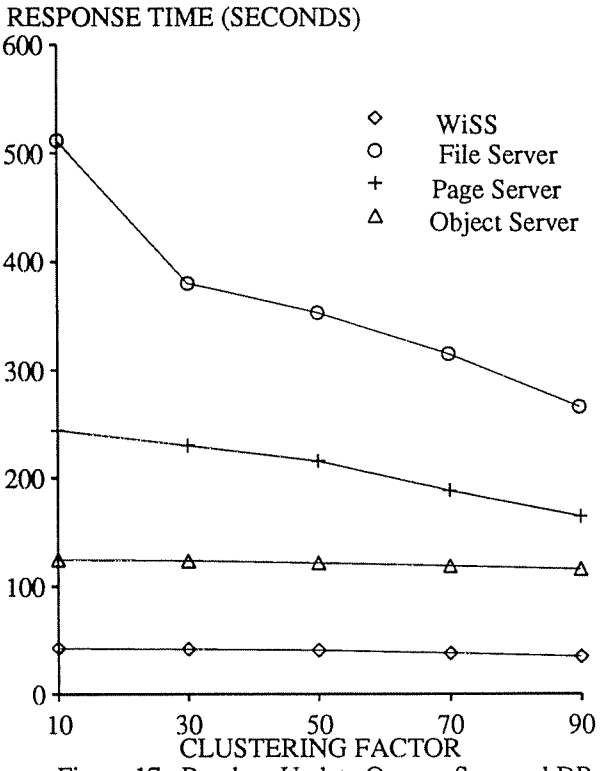


Figure 17: Random Update Query, Smeared DB

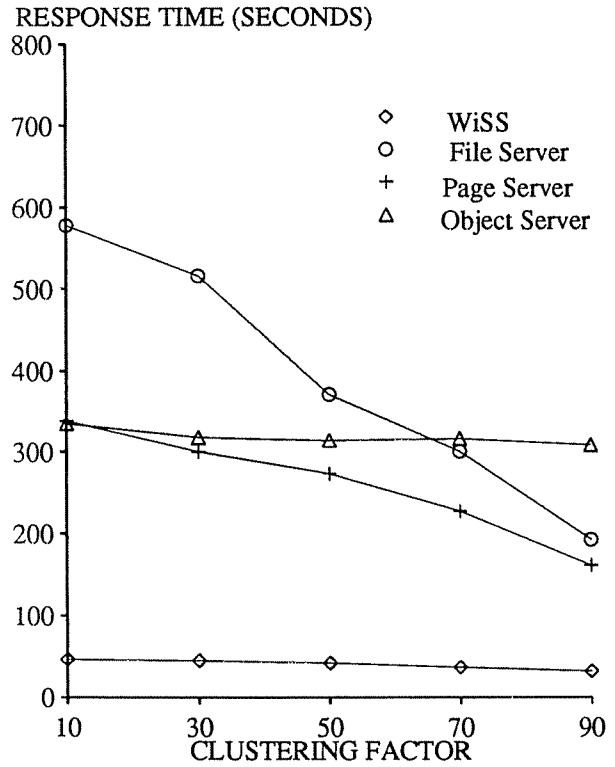


Figure 18: All Queries, Unsmeared DB

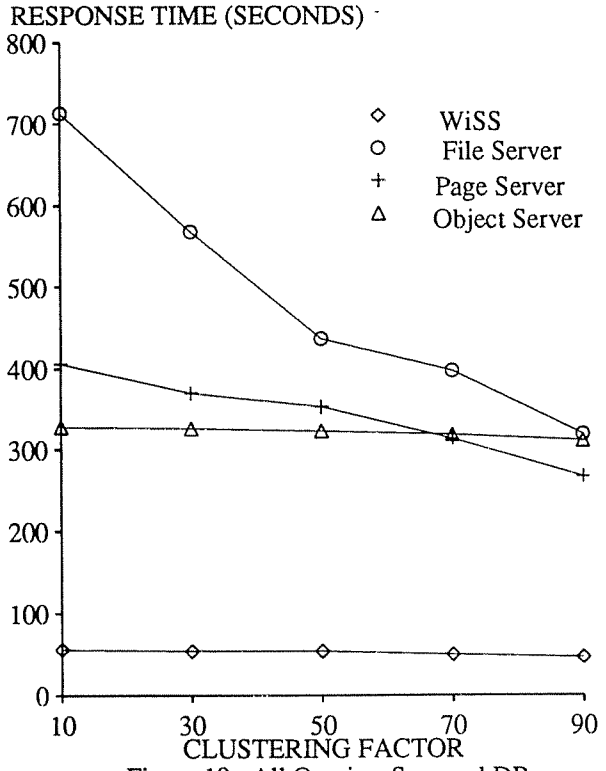


Figure 19: All Queries, Smeared DB

user runs an associative query against a collection in order to select a set of objects to be further manipulated and a second set of objects to be updated. The results obtained for the four different architectures are presented in Figures 18 and 19.

These graphs reveal several interesting results. First, when the database is not smeared, the page-server design outperforms the object server across almost the full range of clustering factors — primarily since the object server performs so poorly on the scan component of the experiment. Again we see that the object server's performance is relatively unaffected by changes in clustering and smearing. As expected from the results for the random read and update queries by themselves, the performance of the page server improves significantly as the clustering factor is increased. Similarly, the performance of the file-server design improves as the clustering factoring is increased because fewer NFS writes are performed. When the database is smeared, the performance of the page and file servers degrades as expected and the object-server design provides slightly superior performance until a clustering factor of 70% is reached. Again, smearing has no significant performance impact on the object-server.

Sensitivity to the Clustering Region Size

After observing the results presented above, we were curious as to the sensitivity of our results to the size of the clustering region. To test this sensitivity, we repeated the four queries using an unsmeared database and with clustering regions of 1 and 9 pages. (Recall that our definition of *clustered* states that an object B is clustered near an object A if B lies within the clustering region surrounding A.) We used only an unsmeared database because the results in Figures 12 through 19 indicate that the primary effect of smearing is to shift the file and page server curves by a nearly constant amount and not to change their fundamental shapes. The most interesting results we obtained are presented in Figures 20 and 21.

Figure 20 presents the execution time of the random read query for the page- and object-server designs for the three sizes of clustering regions. These results demonstrate that the relative performance of the different systems is indeed sensitive to the size of the clustering region. With a clustering region size of 5 pages, the performance of the page server is better than that of the object server only when the clustering factor is above 55%. When the clustering region is shrunk to one page, the crossover point drops to a clustering factor of about 30% and when it is increased to 9 pages the crossover point climbs to about 65%. These results reveal that the page server's performance remains very sensitive to the clustering factor, regardless of the size of the clustering region. However, they also reveal that the performance of the page server will be superior if either the degree of database clustering can be

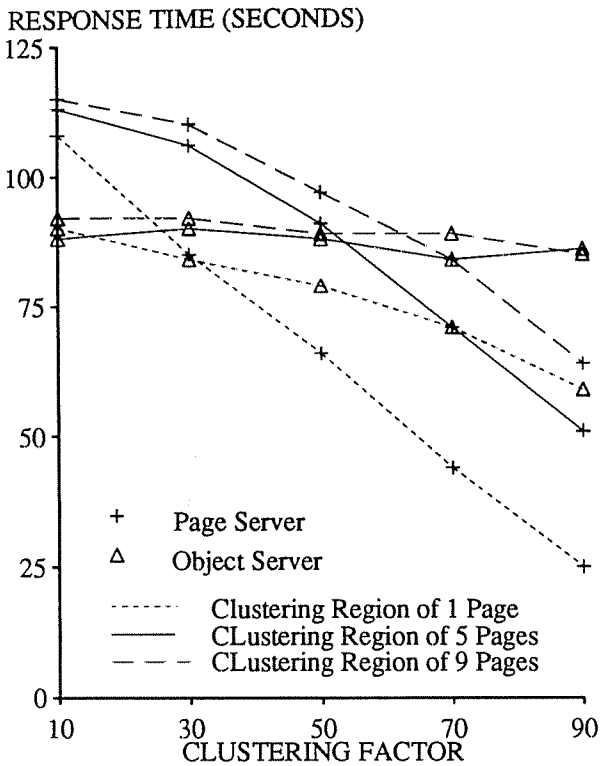


Figure 20: Random Read Query

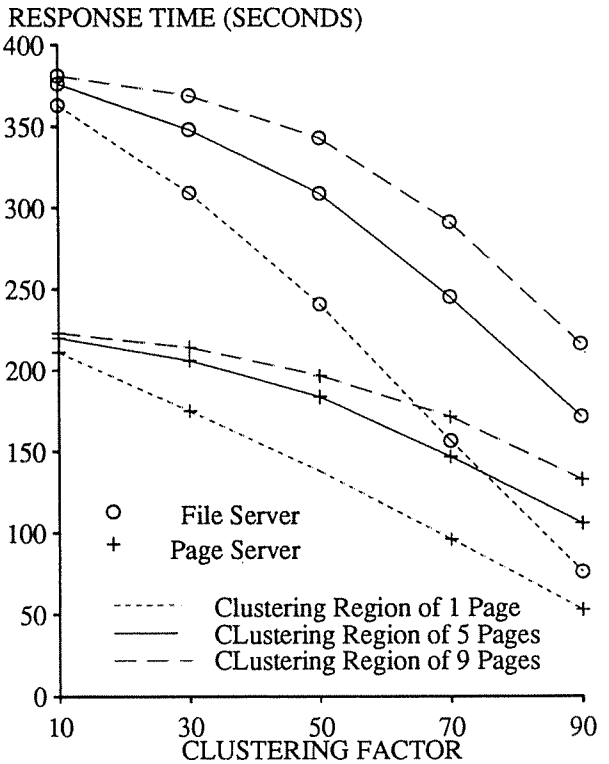


Figure 21: Random Update Query

kept above 60% or if the region of clustering can be restricted to consist of only a few[9] pages.

Figure 21 presents the execution time of the random update query for the page- and file-server architectures (the object server's execution time is a fairly constant 125 seconds). Here changing the size of the clustering region simply shifts the positions of the page- and file-server architectures, and does not change their fundamental shape or spacing. Perhaps the most interesting result is that with a clustering region of 1 page, the page server outperforms the object server when the clustering factor is above 65%.

5.4. Impact of Workstation Buffer Space

For our final set of experiments, we selected two clustering factors (30% and 90%) and varied the size of the buffer pool on the workstation from 50 pages to 300 pages (the size of a set of 1,500 objects).[10] For the page- and object-server designs, the server's buffer pool was fixed at 50 pages. The results are displayed in Figures 22 through 29.

---

[9] Actually, we speculate that the absolute size of the clustering region may not matter. Rather the key is probably the size of the clustering region relative to the size of the workstation's buffer pool.

[10] In the case of the file server prototype, in addition to increasing the size of the buffer pool in 50 page increments, we also increased the "size" of the physical memory of the workstation in 50 page increments.

As expected, the size of workstation buffer pool has no effect on the performance of the sequential scan query. With the object-server design, response time actually increases. The only explanation we can offer is that this increase is an artifact of how the hash table for the object cache was organized. However, the random read and random update queries offer some interesting results. First, while the performance of the page- and file-server architectures both improve as the size of the buffer pool increases, the performance of the object server levels off once the buffer pool has been increased to 150 pages. At this point the cache size is no longer the limiting factor; the objects referenced by the query will all fit in the cache. Instead, the performance of the object server is limited by the cost of transferring objects using RPC. Since the other designs cache full pages in their buffer pools, their performance continues to improve. One would not expect any improvement beyond a buffer pool size of 300 pages.

For the random read query, the relative difference between the page server and file server diminishes as the size of the buffer pool is increased because fewer and fewer remote accesses are necessary. Hence, the performance advantage provided by using NFS decreases.

In the case of the random update query, we observe the following. First, the file server is more sensitive than the page server to the size of the workstation buffer pool. At small buffer sizes, page-write costs are the dominating factor for the file server. However, when the whole database fits into memory, the benefits of using NFS for reads compensate for the more expensive write operations, because no pages are written more than once.

Second, the object server performs much better than the page server when the buffer size is small. At a 30% clustering factor, the buffer size has to be almost the size of the database before the page- and file-server designs outperform the object server. We conclude that more memory is not a reasonable substitute for effective clustering — unless the entire database will fit in main memory. With a 90% clustering factor, the performance of all 3 designs is very close.

For all four queries, we observe that the page server is more sensitive to the size of the workstation buffer pool than to clustering (even considering the various clustering region sizes). Except with a 300-page buffer pool, it was never the case that the pages referenced by a query would fit entirely in the buffer pool, even at a high clustering factor. Large buffer sizes do a better job at compensating for the lack of effective clustering than the opposite (i.e., effective clustering compensating small buffer sizes). For example, with the random read query, a 200-page buffer with a 30% clustering factor performs better than a 50 page buffer with a 90% clustering factor. This behavior is even more noticeable with the random update and all queries tests.

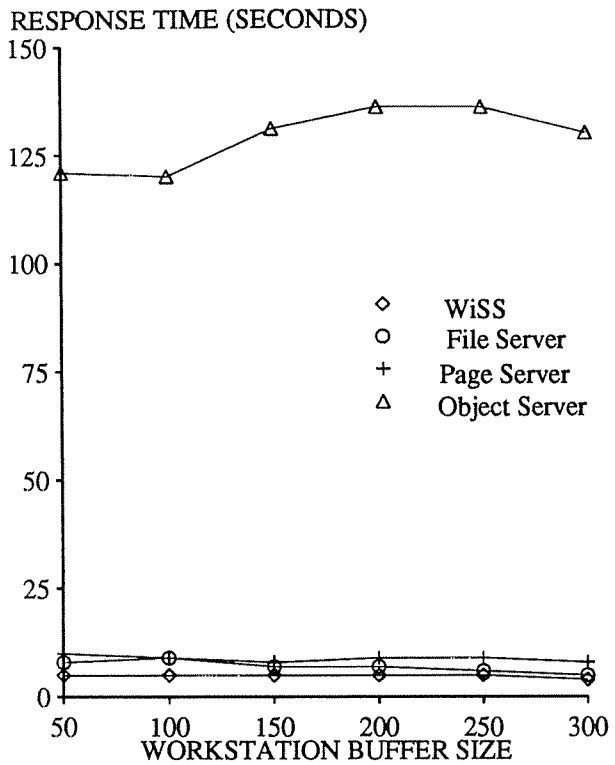


Figure 22: Scan Query, 30% Clustering Factor

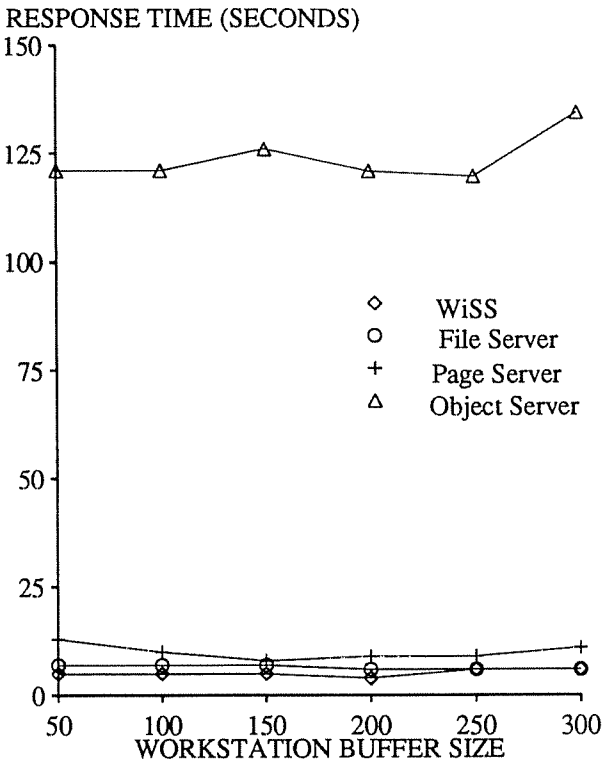


Figure 23: Scan Query, 90% Clustering Factor

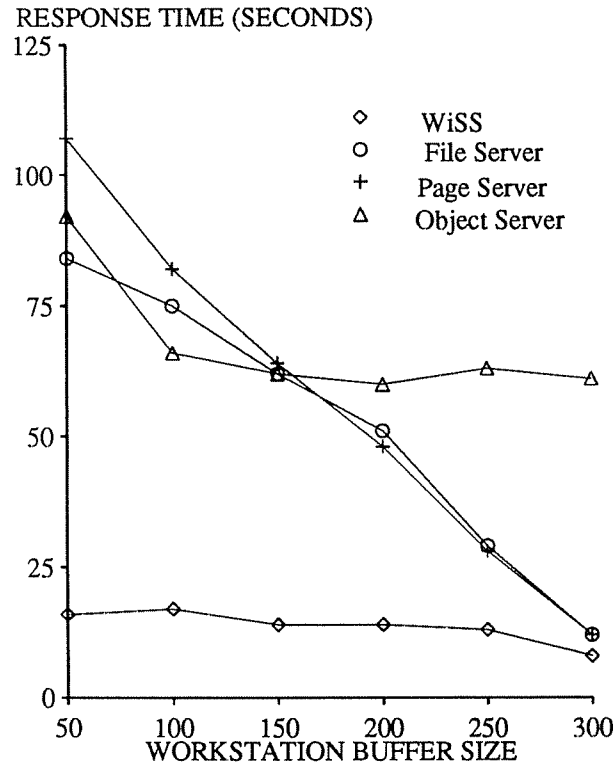


Figure 24: Random Read Query, 30% Clustering Factor

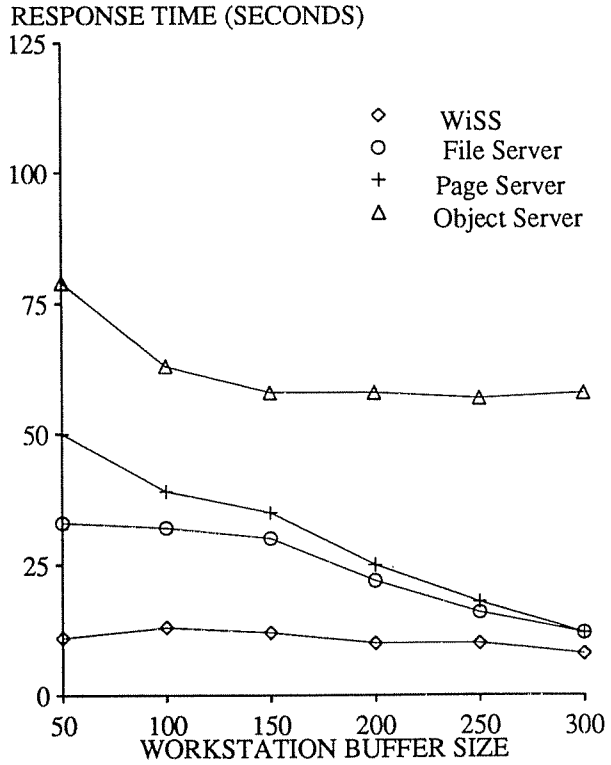


Figure 25: Random Read Query, 90% Clustering Factor

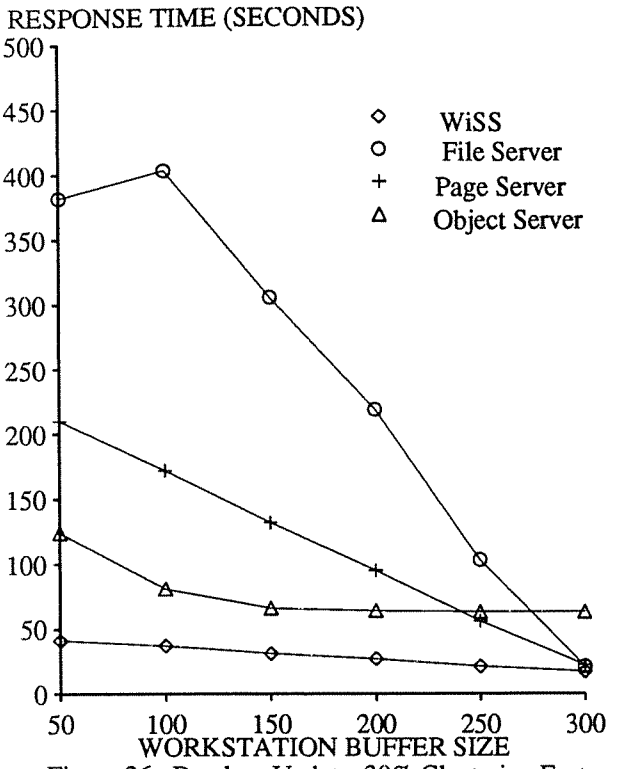


Figure 26: Random Update, 30% Clustering Factor

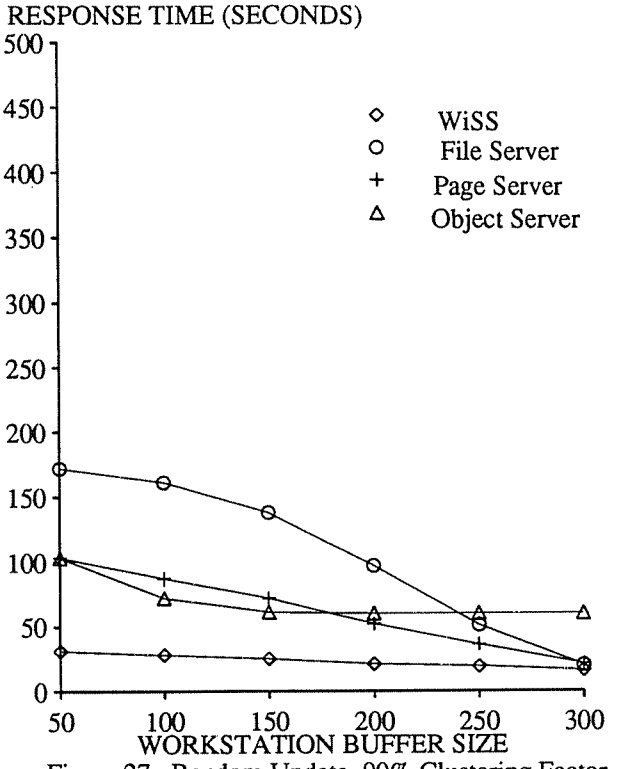


Figure 27: Random Update, 90% Clustering Factor

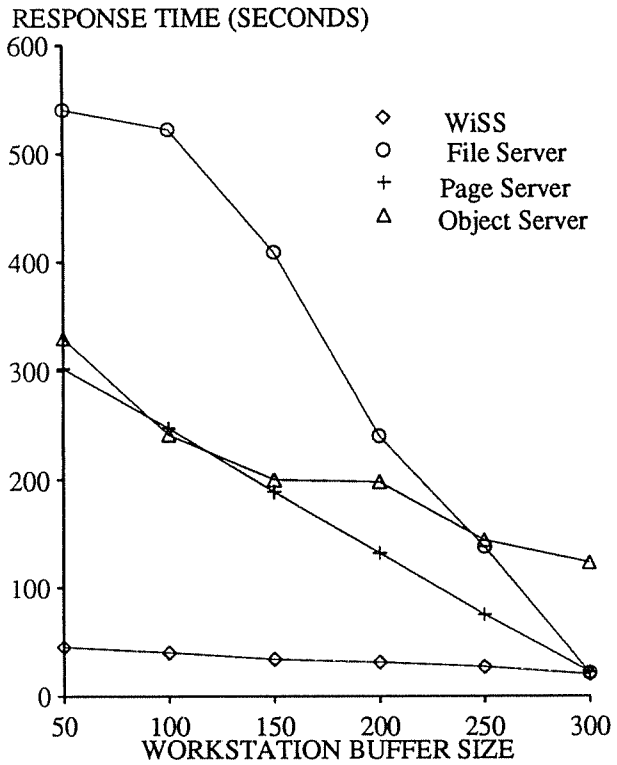


Figure 28: All Queries, 30% Clustering Factor

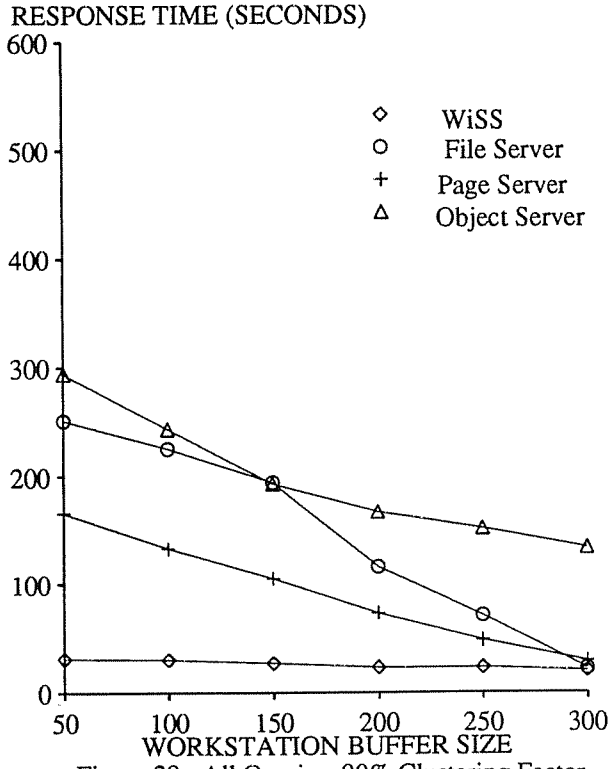


Figure 29: All Queries, 90% Clustering Factor

6. Conclusions

We analyzed three different workstation-server architectures: **object server**, **page server** and **file server**. The object server transfers individual objects between machines, the page server transfers disk pages, and the file server transfers pages using a remote file service mechanism such as NFS.

We analyzed each architecture qualitatively. The main advantage of the page-server and file-server architectures is the simple design of the server, as most of the complexity of the OODBMS is placed on the workstation, an approach that coincides with the current trend towards very powerful workstations. The main drawbacks are that fine-grained concurrency is difficult to implement, that non-two phase locking protocols on indices must involve the workstation, and that objects cannot be manipulated in the server. An object-server architecture implies a complicated server design and suffers from the need to issue an RPC each time an object has to be read from the server. Its main advantages correspond to the page server's limitations: (i) fine-grained concurrency control can be centralized in the server, and (ii) a method that selects a small subset of a large collection can execute on the server, avoiding the movement of the entire collection to the workstation. (This behavior, however, is not easy to implement.)

To compare the performance of the architectures, we built prototypes of all three, using a stripped-down version of WiSS as a starting point. To experiment with sensitivity to data placement and cache sizes, we developed our own object manager benchmark, the "Altair Complex-Object Benchmark."

Our main conclusions of these experiments are as follows:

(1) The object-server architecture is relatively insensitive to clustering. It is sensitive to workstation buffer sizes up to a certain point, at which point its performance is determined by the cost of fetching objects using the RPC mechanism.

(2) The page-server architecture is very sensitive to the size of the workstation's buffer pool and to clustering when traversing or updating complex objects. While the page-server architecture is far superior on sequential scan queries, the object server architecture demonstrates superior performance when the database is poorly clustered or the workstation's buffer pool is very small relative to the size of the database.

(3) The file server's performance is very good when reading pages of the database using NFS, but, writes are slow. On the update query, and in the combination of all three queries, the file server is very sensitive to the size of the workstation buffer pool (even more so than the page server).

Our results are similar to those obtained by Stamos [Stam84] who simulated the behavior of different memory architectures in a non-distributed, single-user object manager, using different strategies for clustering. His conclusions are similar to some of ours, namely, that page buffering is more sensitive to clustering than object buffering, and that object buffering shows better performance than page buffering only when buffer space is very limited relative to the size of the working set of an application. Similar architectural issues also arise in the design of distributed, object-oriented operating systems. In the Comandos [Marq89] system, for example, the default is for the Storage System to respond with single objects when the Virtual Object Memory has an object fault. However, logi-

cal clusters of objects can be declared, and the whole cluster is delivered when any object in it is requested.

The observations above lead us to postulate:

(1) There is no clear winner. A page-server approach seems beneficial if the clustering algorithm is effective and if the workstations have large enough buffer pools. An object-server approach will perform poorly with applications that tend to scan large data sets, but it will perform better than a page server for applications performing updates and running on workstations having small buffer pools.

(2) A file server approach based solely on NFS is ruled out mainly because of its expensive, non buffered writes. Its need for separate messages for locking is also a problem.

(3) A hybrid architecture may be necessary to maximize overall performance. The "naive ideal" would be (i) to read pages through NFS and (ii) to write individual objects back to the server. Each point has a corresponding drawback. For (i), the aforementioned concurrency control problem, and for (ii) the fact that an object cache has to be maintained in the workstation.

While some preliminary studies of different page sizes showed very little impact on the performance of the different designs, we intend to study this issue further. In addition, we must examine multiuser issues more carefully.

Acknowledgements

We would like to thank Jacob Stein for providing us with information on the current architecture used by Gemstone, Mike Carey for his comments on an earlier draft of the paper, and Gilbert Harrus, Jon Kepecs, Rusty Sandberg, and V. Srinivasan for their help in understanding how NFS in V4.0 of SunOs operates.

References

[Ande90] T.L. Anderson, A.H. Berre, M. Mallison, H. Porter, B. Schneider. The hypermodel benchmark, to appear, *Proc. of the 2nd Intl. Conf. on Extending Data Base Technology*, Venice, March 1990.

[Astr76] M.H. Astrahan, et al. System R: A relational database management system, *ACM TODS 1:2*, June 1976.

[Atki89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik. The object-oriented database system manifesto, *Proc. of the First Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1989.

[Banc88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, F. Velez. The design and implementation of O2, an object-oriented database system, In *Advances in Object-Oriented Databases: Proc. of the 2nd Intl. Workshop on Object-Oriented Database Systems*, K. Dittrich ed., Lecture Notes in Computer Science 334, Springer-Verlag, 1988.

[Bern89] G. Bernard, D. Steve. Handling distribution in the O2 system, submitted to the 1990 Intl. Conf. on Distributed Systems.

[Bret89] B. Bretl, et. al., The GemStone data management system. In *Object-Oriented Concepts, Databases and Applications*, W. Kim, F. Lochovsky eds., ACM Press, 1989.

[Care89] M. Carey et al., The Exodus extensible DBMS project: an overview. In [Zdon89].

[Catt88] R. Cattell. Object-oriented DBMS performance measurement, In *Advances in Object-Oriented Databases: Proc. of the 2nd Intl. Workshop on Object-Oriented Database Systems*, K. Dittrich ed., Lecture Notes in Computer Science 334, Springer-Verlag, 1988.

[Chan89] E.E. Chang, R.H. Katz. Exploiting inheritance and structure semantics for effective clustering and

buffering in an object-oriented DBMS, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, Portland, OR, June 89.

[Chou85] H.-T. Chou, D.J. DeWitt, R.H. Katz, A.C. Klug. Design and implementation of the Wisconsin Storage System, *Software—Practice & Experience 15:10*, October 1985.

[Cope84] G. Copeland, D. Maier. Making Smalltalk a database system, *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, June 1984.

[Deux90] O. Deux et al. The story of O2, To appear, *IEEE Transactions on Data and Knowledge Engineering*, March 1990.

[Gray76] J.N. Gray, et al. Granularity of locks and degrees of consistency in a shared data base. In *Modeling in Data Base Management Systems*, G.M. Nijssen, ed., North-Holland, 1976.

[Horn87] M.F. Hornick, S.B. Zdonik. A shared, segmented memory for an object-oriented database, *ACM TOOIS 5:1*, Jan. 1987.

[Kim90] W. Kim, et al. Architecture of the ORION next generation database system, to appear, *IEEE Transactions on Data and Knowledge Engineering, March 1990.*

[Kung81] H. Kung, J. Robinson, On optimistic methods for concurrency control, *ACM TODS 6:2*, June 1981.

[Lind79] B. Lindsay, et. al. Notes on distributed database systems, Technical Report RJ2571, IBM Research Laboratory, San Jose, California, July 1979.

[Marq89] J.A. Marques, P. Guedes. Extending the operating system to support an object-oriented environment, *Proc. OOPSLA '89*, New Orleans, Oct. 89.

[Sun88] *Network Programming Guide*, Sun Microsystems, Part Number: 800-1779-10, May 1988.

[Stam84] J.W. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory, *ACM TOCS 2:2*, May 1984.

[Ston76] M. Stonebraker, et al. The design and implementation of INGRES, *ACM TODS 1:3*, September 1986.

[Vele89] F. Velez, G. Bernard, V. Darnis. The O2 object manager: an overview, *Proc. VLDB XV*, Amsterdam, August 1989.

[Zdon89] S.B. Zdonik, D. Maier. *Readings in Object-Oriented Database Systems*, Introduction: Fundamentals of object-oriented databases, Morgan-Kaufmann, San Mateo, CA, 1990.