BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M1

Performance Improvements in the ObServer Object Server

by
Gregory Delott

Performance Improvements in the ObServer Object Server

by
Gregory Harris Delott
Sc.B., Brown University, 1989

Project

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science
in the Department of Computer Science at Brown University

February, 1991

This project by Gregory Harris Delott is accepted in its present form by the Department of Computer Science in partial fulfillment of the requirements for the Degree of Master of Science.

Date 2/22/91

Stanley B. Zdonik

Stanley B. Zdonik
Advisor

Performance Improvements in the ObServer Object Server

Gregory H. Delott

February 21, 1991

1 Introduction

ObServer is an object server created to oversee the efficient transfer of arbitrarily sized objects from the secondary storage of a centralized server to the memory of a database or other object-oriented application[MFF88][FZE90]. Since ObServer's initial conception, a number of people have worked on it, adding to and modifying various aspects of its functionality. Until recently, however, neither real-time performance nor maintainability was considered a priority, despite the fact that room for improvement in both these areas definitely existed.

The initial and most important goal of this project was to significantly improve ObServer's performance. However, in the process of making these improvements, it became clear that in order to facilitate present and future efforts, a number of items of a software engineering nature needed to be taken care of first. Of primary importance here was understanding and documenting how ObServer works, including how modules were interrelated, what algorithms and structures were used and also what code was unnecessary or unused.

In this paper, I first discuss the tools I used to analyze ObServer's performance. Next, I show how changes other than the ones I made improved ObServer's performance and what the actual effects of these improvements were. Then, after presenting some background information, I give descriptions of the changes I made—first those pertaining to performance, then those for maintainability. Next, I document the effects of the changes I made as well as and the cumulative effects of all the changes since the original benchmark data was generated. Finally, I present my suggestions for possible directions of future research or implementation work along with information on further documentation.

2 Analysis Tools

Analyzing ObServer's performance involved using a profiling tool in conjunction with a set of benchmarking programs. The profiler was used to determine where inordinate amounts of time were being spent while the benchmarks were used to help evaluate the end effects of various changes. It should be noted, however, that data from both the benchmark and the profiler are approximations. Running ObServer involves network communication, and, in some cases, networked file servers.

Both of these services yield variable performance and as a result, there appears to be as much as a 5% time variation in identical tests on very similarly loaded systems.

2.1 The Benchmark

To test the performance of ObServer as changes were being made, I used a set of benchmarking routines devised by Alan Ewald [FZE90]. Of the three benchmarks Ewald described, I chose the one that most closely approximated expected real applications. This set was designed to simulate a computer aided design (CAD) application.

> [The] benchmarks were carried out by creating databases for the application being simulated, then running programs to simulate accesses and updates of the stored objects. Objects were stored in the databases according to the semantics of the application.... Each benchmark did very little computation other than operating on the database. Compute time between database operations was simulated by inserting wait loops executed a random number of times within an upper bound.[FZE90, p. 13]

A number of different parameters were varied to exercise the server. These were:

- Size of the server's segment cache: 1 or 2 MB.
- Number of objects per clustered segment: groups of 100 or 250. "Half of the [3000] objects were clustered into groups of 50 to simulate composite objects.... The remaining 1500 objects were placed sequentially in segments of 100 or 250 objects."[FZE90, p. 17]
- Client's access method: Random or Clustered. Clustered access meant that if the object that was just accessed was part of a composite object, there was a 2/3[1] chance that the next object referenced would be part of the same object (and therefore in the same segment).
- Client-side caching: On or Off. With client side caching on, the client would accept whole segments as the result of individual object requests and would keep all the objects in the segment in its cache.

For more details, see Section 5 of [FZE90].

Unfortunately, the data presented in [FZE90] is not as complete as might be desired. First, while an operation-by-operation analysis of time and other resource uses is given for the program that creates the sample database, this information is missing from the description of the programs that access the data. Only average times to actually access the data are given: timing for supplemental operations, including storing updated data, is lacking. Since typically much more time tends to be spent in accessing and updating data than in generating it, this gives a lopsided view. I suggest what I feel is a more comprehensive benchmark in Section 3.2 below.

Furthermore, it was impossible to fully recreate Ewald's original tests to "fill in the holes". Since the original tests were done, the machines used for running ObServer have been significantly upgraded. Most notable among the changes are much faster NFS[TM2] servers (Sun 4/490s in place of 4/280s) and greatly increased RAM on the workstations (16 MB in place of 8).

[1] [FZE90, p. 18] says "50 percent" but this is wrong.

[2] NFS is a trademark of Sun Microsystems

Finally, the original benchmarks were run with a different random number seed for each test. This unfortunately tends to generate data with an even higher expected margin of error than the 5% mentioned above.

2.2 The Profiler

In order to determine which areas of the server were taking up too much time I used the **gprof** profiler[GKM82]. **gprof** is a statistical profiler which lists, among other things, the approximate amount of time spent in each routine in a program. This tool was critical both in discovering which paths in the server got the heaviest use and in evaluating the specific impact of individual changes as soon as they were made.

However, some unfortunate anomalies do result from using a statistical profiler. As **gprof**'s man file mentions: "It is assumed that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called." Unfortunately, this is not always a valid assumption. One example in which this was particularly true was the case of the *write* system call.

Throughout the course of the changes listed below, the number of invocations of *write* was significantly decreased. A large part of this decrease came from clustering what had previously been groups of *write* calls into single ones to save both *write* and other processing time. As expected, this provided a speedup but it also had a side-effect of increasing the average *write* time. As a result, the routines that decreased the number of times they called *write* appeared to run much faster but routines that had not changed their number of *writes* actually appeared to take up more time than they originally had. This was not only a source of confusion, it also made it difficult to accurately assess the effect of certain changes.

Also, **gprof** was unable to correctly monitor all of the server's time. According to the profiler's output, `main()` and its descendants only accounted for approximately 85% of the server's total time. The other 15% of the time was taken up by such routines as "zword" and "w4cp" which **gprof** claimed were called spontaneously. The fact that these routines sound like they implement zeroing memory and copying data and that *bzero* and *bcopy*, which are used extensively, appear to take no time at all leads one to think that **gprof** was unable to correctly trace these system calls.

Finally, **gprof** only monitors time spent in user mode. This is unfortunate since a large percentage of ObServer's time is spent in system mode, reading from and writing to disks and ports. Thus the profiler showed only a minimal decrease in run time when the file system was switched from NFS to a local disk. For example, in one test, **gprof** claimed that the time the server spent running decreased from 92 to 85 seconds when the files were moved from NFS to a local disk. In fact, the amount of time the client spent waiting for the server decreased from 2120 seconds to 614. Thus, even taking into account the fact that the server does spend some time doing things when the client is not waiting for it, is it still quite possible that as much as 4/5 of all the server's time is not being profiled when using a local disk. Though this is not as bad as the 19/20 that it is missing when using NFS, it is still very significant and hints that the importance of *read* and *write* calls is vastly understated.

3 Initial Performance

As was mentioned above, the initial set of ObServer performance benchmarks was done by Ewald in [FZE90]. However, since these benchmarks were taken, ObServer's performance has been significantly improved by changes to the the conditions under which ObServer is run. Therefore, before attempting to evaluate the effects of the changes I made, it is first necessary to account for the performance improvements for which I am not responsible.

3.1 System Environment Changes

The first significant improvement to the performance of ObServer came, easily enough, from changes to the environment in which it was run. These changes, which involved both hardware and software, are shown in Table 1 below.

Table 1: Comparison of the Original and Present Benchmarking Environments

Item	*Original Environment*	*Present Environment*
NFS Servers	4 Sun 4/280s	2 Sun 4/490s
Local Machine RAM	8 MB	16 MB
Swap Space	32 MB	140 MB
Block Size	1024 Bytes	8192 Bytes
Page Size	1024 Bytes	4096 Bytes
Operating Systems	SunOS 4.0	SunOS 4.1
Compiler Optimization	NONE	Level 2

It should be noted that the new NFS servers are significantly faster than the old ones. In addition, with the new operating system came a new version of the compiler. Since enough other factors were being changed, and direct comparison would be impossible anyway, it was decided that there was no reason not to turn compiler optimization on.

Aside from the factors listed above, no changes were made in the environment. The benchmarks were still performed on *Sun SPARCstation 1s*^{TM}[3] and the database files were still stored via NFS. The NFS servers had 32 MB of RAM. Additionally, the benchmark tests were again run on lightly loaded systems and client and server applications resided on the same machine.

As shown in Tables 2 and 3, the effects of these changes to ObServer's environment were quite significant. Table 2 shows statistics gathered during the creation of sample databases. The "Objects" entries in the column headings refer to the number of objects in each segment with clustered objects. The "Time" row gives the total time (in seconds) the client spent waiting for the server. Though this should include time spent waiting for all the routines, it perforce omits a few minor ones that were not listed in [FZE90]. The data in the other four rows is also taken over all the previously mentioned server routines; for the exact definitions of what they measure, see *getrusage(2)*. The server-side segment cache was 1 MB in all cases. As can be seen, the improvement in the machine configuration resulted in drastically decreased quantities of page faults and file-system reads. These and other other factors account for an average overall speedup of 22%.

[3] SPARCstation is a trademark of Sun Microsystems

Table 2: Database Creation Statistics—Effects of Environment Changes

Item	*Original Environment*		*Present Environment*		*Average*
	100 *Objects*	250 *Objects*	100 *Objects*	250 *Objects*	*Improvement*
Time	177	245	162	169	22%
Minor Page Faults	413	408	270	286	32%
Major Page Faults	928	925	208	211	77%
File System Reads	90	65	51	52	34%
File System Writes	653	627	606	602	6%

Table 3 lists even more telling statistics: the average amounts of time the client was idle while waiting for the server to return a requested object. The times given (again, in seconds) are only for the case where there is no client-side segment cache and accesses to the database are clustered. This particular case was picked more-or-less randomly but it should provide a fair idea as to how access time increased in general. The "Cache Size" is the size of the server-side segment cache; "Objects" is again the number of objects in each segment with clustered objects. It should be noted that the numbers given for the "Original Environment" are approximate: they had to be read off a bar graph in [FZE90]. Overall, the object access times were two to three times faster than in [FZE90].

Table 3: Average Object Access Times—Effects of Environment Changes

Cache Size	*Objects*	*Original Environment*	*Present Environment*	*Improvement*
2 MB	100	1.80	0.66	63%
	250	1.90	0.60	68%
1 MB	100	1.80	0.88	51%
	250	2.15	1.02	53%

3.2 NFS vs. A Local Disk

Even more significant was the speedup obtained by using a local disk for database files in place of NFS. Considering the ObServer is, after all, a database program, the fact that a (fast) local disk made a tremendous difference in its run time is not at all surprising. In Table 4 below, a number of different operations are compared. They are as follows:

- Create Database: The total time necessary to create a database as described for Table 2 above. Here, all the routines are accounted for so the times are slightly higher than shown in Table 2.
- Create & Access: The total time spent running the database creation program and then three concurrent access programs.
- Average Access: The average amount of time spent to access an object. The data were calculated over a set of 3003 total requests.

- Average Commit: The average amount of time spent to commit an object. The data were calculated over a set of about 1000 total requests.

The first two items above should give good general ideas as to overall performance for a task. The latter two are critical tasks for the server that happen very frequently. Together, these four items provide a good benchmark for the performance of the server.

All the numbers reflect the amount of time the clients were idle while waiting for the server to respond. For "Create Database", times are given for both 100- and 250-object segments. For the other three tests, average times are given. To generate the data in the "No Cache, Clustered Access" rows, clients had no segment cache and made clustered accesses. The results presented are averages of times taken for servers that stored both 100- and 250-object segments using both 1 and 2 MB segment caches. To generate the data in the "250 Objects, 2 MB" rows, servers stored 250-object segments using 2 MB segment caches. The results presented are averages of times taken for clients both with and without caches making both clustered and random accesses. This provides a cross-section of results over the four variable parameters.

Table 4: NFS vs. Local Disk Times

Test	*Parameter(s)*	*NFS*	*Local Disk*	*Improvement*
Create Database	100 Objects	165.2	52.2	68%
	250 Objects	172.9	49.4	71%
Create & Access	No Cache, Clustered Access	3178.1	575.9	82%
	250 Objects, 2 MB	2576.5	607.5	76%
Average Access	No Cache, Clustered Access	0.573	0.102	82%
	250 Objects, 2 MB	0.658	0.091	86%
Average Commit	No Cache, Clustered Access	0.787	0.201	75%
	250 Objects, 2 MB	0.540	0.219	59%

Note that all of the above results are between three and seven times faster than when NFS was used. This is a tremendous improvement and clearly shows that ObServer should always be run on a local disk. Also note that without this change, any other improvement that did not directly involve disk accesses would have a negligible effect since it would have been overshadowed by disk access time.

4 Background Information

In order to appreciate a number of the changes presented below, it is important first to have a fairly detailed understanding of three significant aspects of ObServer. These aspects are:

1) ObServer's overall structure.
2) The server's checkpointing and recovery system.
3) How object mapping is done.

4.1 ObServer Structure Overview

ObServer source code is divided into four major groups. Each group has a three letter prefix and the names of all the files in that group begin with that prefix. The groups are:

BINDER	("BND")	—	Code and information specific to the Binder and Daemon.
COMMON	("CMN")	—	Code and information common to all the groups.
CLIENT	("CNT")	—	Code and information specific to the Client.
SERVER	("SVR")	—	Code and information specific to the Server.

Within each group are a number of modules, each with its own specific domain. Almost none of the work in this project was done on modules in either the BINDER or CLIENT groups and they are for the most part not mentioned again in this paper. Only a few of the COMMON group modules were significantly modified and where these modules are mentioned below, it is made explicit that they are part of the COMMON group. Otherwise, modules mentioned below can be assumed to be part of the SERVER group. For a complete list of the SERVER group modules, see Appendix A below.

4.2 Checkpointing and Recovery

ObServer's crash recovery system is a simple but effective one. Each request that comes into the server is logged in a *recovery log*. Periodically (or when requested to by a client), all the server's files *except the database file itself* are checkpointed and the log is emptied. Thus, if the server crashes or is killed, it can easily be recovered to the state it was in just before the crash. To do this, the supplementary files have to be restored to their checkpointed versions and then the requests are "replayed" from the log.

Checkpointing of supplementary files is done in one of two ways. For some modules (CF, DOT, OLT, SMF and SVR), a "backup" style of checkpointing is done. At checkpoint time, the entire contents of that module's files are copied to files that begin with the same name but have ".bak" appended. For the other file-using modules (CAT, DEAD, LOCK, NUM, SGM and TRANS), a "shadow" naming convention is used. In their case, when it is time to do a checkpoint, they write out all their data to files with either a ".0" or ".1" extension, depending on the state of a master switch. This is possible since they keep all their data in main memory. The "File Memory Management" item in Section 8.2 below suggests a way to improve the efficiency of this process.

The database file itself, on the other hand, is never really checkpointed at all. Rather, it is always kept in a state such that restoring the supplementary files automatically causes the database to appear to revert to its state as of the last checkpoint. To understand how this works, it is necessary first to understand how segments are created and maintained.

When the Segment Create routine (`SEGcreate()`) creates a segment, it first obtains a new Virtual Segment Identifier (Virtual SID) for the segment. This Virtual SID is immutable and is thought of as the segment's ID at all levels above the Segment module. Then the Segment Create routine calls the Secondary Storage Create routine (`SECScreate()`) to allocate the new segment an actual location in the database and a new Real SID to refer to this location. The Secondary Storage Create routine stores this mapping from the Real SID to the database location in the Segment Pointer Table (SPT) at the front of the database file. The mapping from the Virtual SID to the Real SID is stored in the Segment Mapping Table.

When the Segment Write routine (`SEGwrite()`) routine is called to write an updated segment to the database, it does one of two things. If the segment ***has not*** already been written to the database file since the most recent checkpoint, the Segment Write routine calls the Secondary Storage Create routine to create a new Real SID and a new location for the modified segment. This is done to ensure that the Real SID as of the most recent checkpoint always points to the same memory location and that the data at this location is never tampered with. As is the case with segment creation, the SPT holds the mapping from the new Real SID to segment's new location and the SMT holds the new Virtual to Real SID mapping. The segment is then written to its new location and the old Real SID is marked as unused—though it and the database blocks to which it points are not available to be reused until after the next checkpoint. Otherwise, if the segment ***has*** already been written to the database file since the most recent checkpoint, it is written back to this location if it fits. If it does not fit, it is stored somewhere else and the Real SID's entry in the SPT is updated.

At checkpoint time, the Virtual to Real SID mappings in the SMT are transferred to the SMF and this file is backed up. Additionally, database file blocks pointed to by deleted Real SIDs are then considered free, as are the deleted Real SIDs. If the server crashes and needs to recover, the backed up version of the SMF is restored. This has the Virtual to Real SID mappings as of the most recent checkpoint and since both the Real SIDs and the memory to which they point remained untouched, these mappings are still valid. The Virtual to Real SID mappings created since the last checkpoint will already have been lost with the SMT and any mappings that were in the SPT for new Real SIDs are ignored. Thus the database file is thus automatically "restored".

4.3 Object Mapping

A significant fraction of the ObServer code is devoted to mapping the External Unique (Object) Identifiers (External UIDs) that the client holds to the actual data to which they refer. To appreciate the motivation behind some of the major changes and suggestions below, it is first necessary to understand the mechanisms of this translation. Note that all forms of UIDs are 64-bit values.

The Object Mapping process can be divided into three major steps. These are:

1) Translating the External UID into (an) Internal UID(s).
2) Loading the segment(s) specified by the Internal UID(s) into main memory.
3) Locating the desired object(s) within the segment(s).

A detailed description of each step follows[4]. This mapping is also shown graphically in Figure 1.

Step 1 — Getting Internal UIDs

The External UID—which is a {*File*, *Index*} pair—is looked up in the Object Location Table (OLT). This is a simple matter of reading in the entry at index *Index* in OLT file *File*. Based on the two high-order bits of the result, this is interpreted as one of three things:

- A tombstone. This means the object in question has been deleted and the lookup ends here.
- An Internal UID. This is a {*Virtual SID*, *Object Index*} pair whose use is described below.

[4]Section 4.2 in [HZ87] also describes this process, including a diagram (Figure 3). However, although this description gives more detail in some respects, it is not completely accurate in others.

- A Duplicate Object Table (DOT) UID. This is used for objects that have been duplicated or moved since they were originally created. The structure of a DOT UID is a {*File*, *Index*} pair which must be looked up in the DOT to produce a set of Internal UIDs. DOT UIDs are looked up in the same matter as External UIDs are, save that they map to fixed-sized groups of Internal UIDs, with optional pointers to additional groups, if needed.

Assuming the desired object had not been deleted, this process thus yields an Internal UID or possibly a set of them.

Step 2 — Loading Segments

For each Internal UID thus obtained, the Segment (SEG) Open routine (`SEGopen()`) must be sure that the segment with the specified Virtual SID has been "opened" and is in main memory. To do this, it first consults the Main memory Segment Table (MST) to see if the Virtual SID is listed. If so, the segment is already in memory and the rest of this step is skipped. Otherwise the procedure for reading in a segment is as follows.

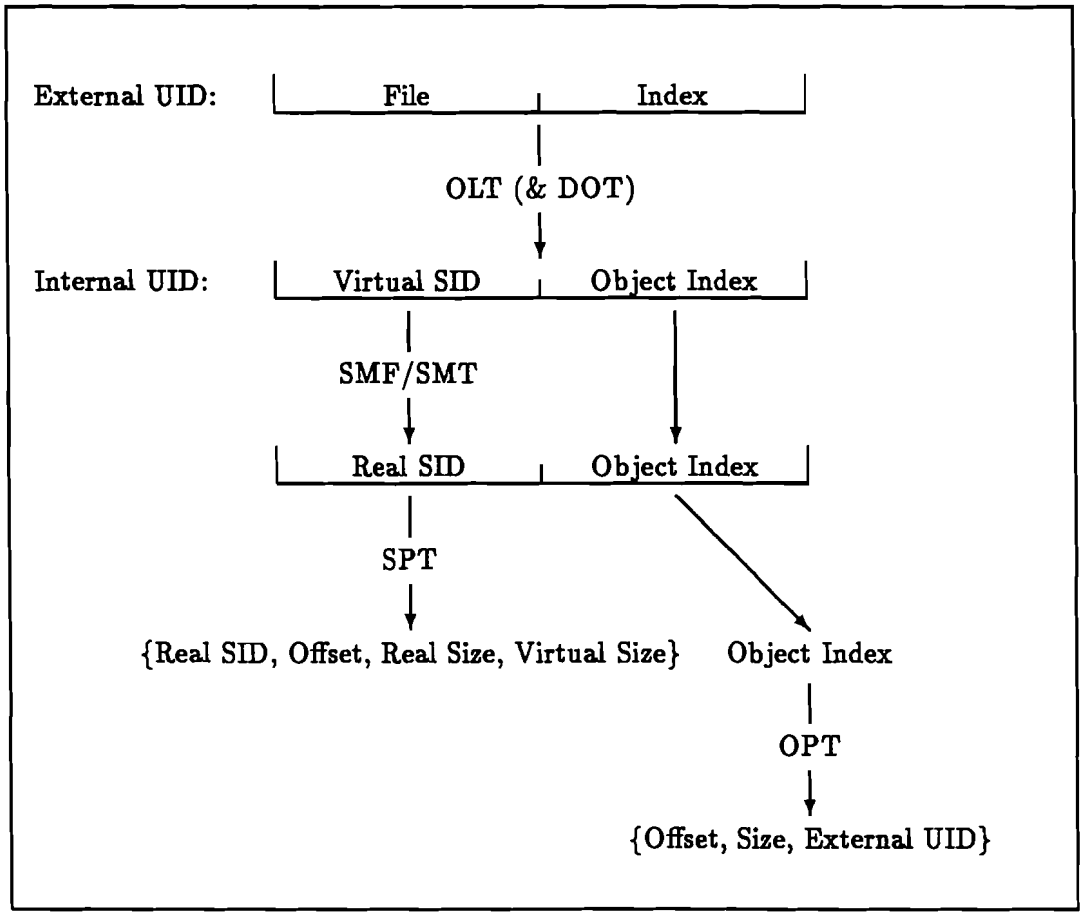
First, the Virtual SID must be mapped to a Real SID. This is necessary because the database stores segments according to Real SID only. This is done in one of two ways. If the segment has been altered since the most recent checkpoint, a new Virtual to Real SID mapping will have been entered in the Segment Mapping Table (SMT) which is kept entirely online. Thus the SMT is first searched for the desired Virtual SID. Otherwise, the lookup must be accomplished by consulting the Segment Mapping File (SMF) which is kept on disk and cached where possible. In either case, this process yields a {*Real SID*, *Object Index*} pair.

This Real SID must then be located in the database file. The Secondary Storage (SECS) Read routine (`SECSread()`) accomplishes this by consulting an extendible hash table of segments, located at the front of the database file itself. By traversing this tree, the appropriate entry in the Segment Pointer Table can be found. This SPT Entry (SPTE) will have the mapping from the Real SID to a {*Real Segment ID*, *Offset*, *Real Size*, *Virtual Size*} quadruple. The *Offset* and *Real Size* are used to locate the segment in the database file, the *Virtual Size* is how big the segment was initially allocated to be and the *Real Segment ID* is redundant. Once the segment is located, it is read in and stored in the MST.

Step 3 — Locating Objects

Finally, the Single Object routines in the Segment module (`SEG_install_single_obj()`, `SEG_update_single_obj()` and `SEG_delete_single_obj()`) use the Object Index from the each Internal UID to locate objects within the segments. This is a simple matter of finding the entry at index *Object Index* in the Object Pointer Table (OPT). This OPT Entry (OPTE) contains a {*Offset*, *Size*, *External UID*} triple. The *Offset* and *Size* are used to locate the object in the segment and the *External UID* is redundant.

Figure 1: Current Object Mapping Procedure



5 Performance Enhancements

To enhance the performance of ObServer, a number of design and implementation changes were made. These changes mostly involved reimplementing key data structures and algorithms in heavily used areas of the code. This section gives brief descriptions of these changes, presenting the motivations for and the approximate effects of each change.

There are, however, two items to keep in mind when examining the documented effects. First, as was mentioned in Section 2.2 above, the impact of individual changes was sometimes difficult to determine both because of the statistical nature of the **gprof** profiler and because its failure to take time spent in system mode into account. Second, the percent improvements given below are based on a single run of the benchmark test using a single set of parameters[5]. Nevertheless, they should provide a general measure of the relative usefulness of each change.

5.1 OLT and SMF Caches

The Object Location Table and Segment Mapping File modules both use caches to provide quicker access to the potentially very large amounts of data they store in files. Previously, this caching was provided through separate Cache (CACHE) and Segment Mapping File Cache (SMFC) modules for the OLT and SMF respectively. However, there were a number of problems with the CACHE and SMFC modules.

The first problems were from a software engineering standpoint. Since the basic purpose of both modules was to provide LRU caching for other modules in fixed-sized blocks, there were large portions of identical or nearly-identical code. Although the interfaces and the cache search routines were slightly different, the buffer management and paging code were identical.

However, more importantly, they were very inefficient. The main cause of this inefficiency was the fact that the caches were too small to be useful. Considering that the *SPARCstations* now have 16 MB of RAM and that the time they take to read or write any amount of data up to 8 KB is nearly constant, using caches of 120 or 60 byte blocks (as the CACHE and SMFC modules respectively used) was a mistake. The frequent need to flush and reload the caches caused frequent *fseeks*, *fwrites* and *freads*. For the OLT, this problem was compounded by the fact that each OLT file contained only 1000 entries. This meant that non-clustered accesses usually also caused the current OLT file to be closed and a different one to be opened. Additionally, as a minor matter, in implementing an LRU paging algorithm, the CACHE and SMFC modules used actual timestamps to determine which block of data was oldest. During the course of running the benchmark, this resulted in thousands of calls to *gettimeofday*.

In order to correct the above problems, it was decided to completely redo the caching. First, the CACHE and SMFC modules were removed. Their purposes were not unique enough to merit their existence as separate modules. To replace them, a few cache-management macros were added to the GLOBAL module in the COMMON group. What little unique material they did contain was put directly into the OLT and SMF modules so that they could do the caching themselves. Keeping the majority of the common code in a common place not only makes maintainence easier, it also cut the quantity of code way down. The combined sizes of the old CACHE, OLT, SMF and SMFC modules was over 67 KB; now, the OLT and SMF only take up about 32 KB between them.

[5] 2 MB Server segment cache, 250 objects per clustered segment, Clustered client accesses, No client-side caching.

But of greater interest is the fact that the new implementations deal with the above efficiency issues. The cache buffers in each module are now each 8 KB in size. This takes care of the first two problems. And, with the help of the new "cache" macros which implement doubly linked circular lists, ordered by the most recent access, they no longer need to call *gettimeofday*. This proved to be the single most-important change made, cutting out over 15% of all the server's time. It should be noted that since the OLT is very heavily used and the SMF is very lightly used, this reduction occurred almost entirely in the OLT.

5.2 Message Handling

The server sends a number of messages to the client. In response to any given client request, the server may send anywhere from zero to half a dozen or more informational messages, as well as a final signal that the desired request has been processed. Previously, each of these messages would be sent via a separate call to the Communication module's Write to Port routine, `COMMwrite_to_port()` in the COMMON group). Each of these calls would involve processing as well as at least one call to the system routine *write*, which requires crossing the user/system boundary and is expensive. Since all the informational messages were being sent out one after the other and the operation completion signal was coming out only a little later, it was decided to bundle these items together.

This decision required significantly rewriting the Server Message Module (SMM) Send (`SMMsend()`) routine. First, instead of making one call to the Write to Port routine per type of message to go out, it now concatenates all the outgoing messages into a single buffer and sends this out. This yields a moderate speedup at the expense of some data copying. See the "Writing to Ports" suggestionwritev in Section 8.2 below for a way to eliminate this copying.

Secondly, the Send routine now takes care of sending out the operation completion signal itself whenever possible. Previously, the SMM's Procedure Return routine (`SMMproc_return()`) would always be called to do this. Now, the Procedure Return routine is only called when the Send routine is not, i.e. when:

1) The Client or Transaction ID of the incoming message is determined to be invalid.
2) The requested operation cannot be performed or has failed for some reason.
3) The operation normally returns no informational messages and thus never calls the Send routine.

This has had the effect of halving the number of calls to the Write to Port routine and yielding an improvement of over 4%.

A small change was also made in the handling of incoming messages. Previously, a buffer would be *malloc*ed every time a new message would come in and *free*d once it was done being processed. Now, the incoming message buffer (now called `SVRmsg_block`) is kept around between messages. Its size is increased with *realloc* when necessary. This prevention of unnecessary *malloc*/*free* pairs with each new message saves about 0.25%.

5.3 Queuing Module

The Queuing module (QUE) is used to maintain enlargeable data buffers which are used by other modules. Originally, the data in each queue was kept in individual nodes in a linked list. These nodes could be added to either end of the queue, were individually dequeueable and could also be compressed into one large node to be dequeued. This was a good general-purpose utility, but it

was unnecessarily powerful.

Careful inspection of the actual usage of the queues revealed that elements were never needed singly, that a large amount of copying was being done to compress them and that only single, fixed-size headers were ever added to the front of the queues. Therefore, they were reimplemented as byte streams with control information. This control information includes the number of elements in the list, the size of the list, the size of the buffer and whether or not there is a header. Now when elements are added to a queue, they are simply concatenated onto the end of a buffer, rather than being made into nodes and linked into a list. The main advantages of this method are the removals of both the *malloc/free* pairs associated with adding and deleting nodes and the elimination of unnecessary *bcopy* calls. All in all, the change accounted for a speedup of about 3%.

5.4 Recovery Log

The server usually runs in NORMAL mode. In this mode, it receives incoming requests, processes them, logs them to a *recovery log* file and then sends the client its response[6]. However, if for some reason the server has crashed or was killed, it is restarted in RECOVERY mode. In this mode, the server ignores client requests until it has "replayed" all the requests logged in the recovery log and brought itself back to the state it was in just before it went down. Then it switches back to NORMAL mode.

Each time the server was to write an incoming request to the recovery log, it would first do an *fseek* to the end of the file. Similarly, when operating in RECOVERY mode, it would also do an *fseek* to the correct point in the file to read the next request to be replayed. This was completely unnecessary. Storing and retrieving requests are both serial processes and, as described above, are never interspersed. Thus the file pointer never needed to be explicitly changed. Removing the *fseek*s accounted for a savings of about 1.25% in the writing routine and would presumably have brought about a similar improvement in the read routine, had recovery been part of the benchmark.

5.5 COMMON Group Code

A few changes that had significant effects on the performance of the server were implemented on code in the common group. For the most part they were trivial to implement.

- (Communication module) Since large amounts of data were being transferred through the sockets, it was decided that the send and receive buffers should be as large as possible. So, these buffers were increased from their default 4 KB to the maximum allowed, 32 KB. This caused a small speedup.
- (Communication module) Previously, *select* was being called before each attempt to read or write data on a port, even if no timeout was specified for the operation. This was pointless since calling *select* on a single port with a NULL timeout blocks until the port is ready, which is what *read* and *write* do anyway. This has been fixed.

[6]Previously, this process was reversed: the request would first be logged and then it would be processed. However, this presented a problem. If this request had somehow caused the server to die, it would be in the recovery log, and would be replayed when the server came back up, quite possibly "killing" the server again. Under the current system, requests are only logged if the server is able to successfully process them.

Also, for each message that was to be received or sent out, *select* was being called before each individual *read* or *write*. Now, *select* is only called the first time, to be sure the port is ready. After that, it is assumed that the port has not stopped transmitting or receiving in the middle of a message. In the unlikely case where this has happened, the alarm clock (which goes off every 30 seconds) eventually terminates the *read* or *write*. The result of these two changes is a savings of about 2.5%.

- (Global module) There are three macros in this module for setting, unsetting and checking bits in (64-bit) masks. Previously, these macros called *abs* to make sure the bit to be checked was a positive number. This was done despite the fact that the macros were always called with positive operands. Since these macros were called a total of about 1.18 million times throughout the course of the benchmark, the trivial action of removing the call to *abs* produced a savings of about 1.25%.

- (System module) Previously, the macro provided for calling the *malloc* system call would always call *bzero* so that the memory area returned would be zero-filled. This was unnecessary in a number of cases. Now, a second macro is provided (and frequently used) which does not call *bzero*. Through the course of the benchmark, this removed about 70 thousand of the initial 98 thousand *bzero* calls. The profiler never showed *bzero* to take any time in the first place but the "zword" routine (which is believed to be called by *bzero*) accounted for about 2.25% less time after the change was made.

5.6 Bug Fixes

The enhancements to ObServer also include two actual bug fixes. The first concerns the Number (NUM) module which is responsible for issuing External UID numbers. A previous change in the way External UIDs were numbered had caused a programming bug to manifest itself as an off-by-one error in the number of UIDs that were given out in response to client requests. The bug was fixed and some redundant `#defines` that had helped cause the problem were removed.

The second problem was in the Segment Mapping File Cache (SMFC) module. A bug in the code that selected which part of the SMF was to be removed had caused the cache not to contain what it was supposed to. This problem would cause the server to hang when the SMFC was used heavily enough. This bug was first fixed and then rendered moot when the SMFC module was removed (as noted in Section 5.1 above).

6 Usability and Maintainability Changes

There were also a number of changes that were made to ObServer that were unrelated to performance. Most of these changes were of a software engineering nature. Descriptions of the significant changes are listed below. For a list of additional, less critical changes, see the file "changes".

6.1 Deleted Files

A large number of unnecessary files were deleted. Among these were all the files for three separate modules. None of the material in these modules was being used. The modules were as follows:

- The Operation Filter (OF) module in the SERVER group. In theory, this module was used to allow the client to "filter out" certain server operations, preventing them from being called in the future. The `SVRset_op_filter()` server routine was provided for selecting these operations. However, this routine (which does not appear to be documented anywhere) was not fully implemented and the module itself could never be accessed. For this reason, the module was deleted.

- The Lock Filter (LF) module in the COMMON group. Like to the OF module above, this module was designed to allow the client to "filter out" certain lock combinations from being requested. Through the `SVRset_lock_filter()` server routine, the client could specify what kind of environment (cooperative, strict, etc.) was to be used. But, like the OF module, this functionality was undocumented and not fully implemented.

- The Object Hash Table (OHT) module in the COMMON group. It is unclear how this module was to be used.

Both the OF and LF concepts appear to be useful ones. But until a decision is made to fully document and implement them, it seems best to keep them out of the way. If they are ever needed, they can be restored via RCS[WFT82]. For a complete list of deleted files, see the file "deleted_files".

6.2 Other Deletions

The ObServer code had many unused constants, variables, macros and routines. Additionally, there were a number of procedure declarations and `#include` lines that had become unnecessary for various reasons. Furthermore, there were routines which had been commented out, were never fully implemented or were otherwise unusable. This material not only made ObServer code harder than necessary to work with, it also added excess bulk to the executable files. So, as much as possible, the above items were deleted. As with the deleted files, RCS can be used to retrieve any of these items in the unlikely case that they turn out to be needed.

6.3 Reformatting

All the files were reformatted. Although there was a standard format evident for the original ObServer source files, these standards had deteriorated with the work of subsequent developers. Perhaps the most obvious change is that all the lines are now less than 80 characters wide except where quoted strings make this necessary. Also, a number of general formatting rules were applied to make the code more consistent and readable. Most of these changes were accomplished through the use of the formatting programs **indent** and **cb**. See the file "indent" for a list of options used with **indent** and their meanings.

6.4 Renaming

Previously, a number of client files had included various server header files (especially "SVRproc_defs.h") and it was often necessary for client programs to link a Server object file ("SVRsvr_utils.o") into their images. This goes against the preferred structure of ObServer which specifies that only COMMON group files are to be used in this manner. To fix this problem, "SVRproc_defs.h" was renamed to "CMNprocs.h" and "SVRsvr_utils.c" became "CMNprocs.c".

This change is also consistent with the contents of the two files. Now, no references to Server files are necessary for the client.

6.5 Rewriting the Scheduler Module

The entire Scheduler module (SCHED) was rewritten. The original data structure used by this module was a circular linked list. In this list were intervals between which the alarm should be signaled and lists of routines to be called when an alarm was signaled. This was messy and hard to maintain. The new data structure is simply an array of routines to be called and their frequencies (`SCHED_array`). The code to deal with this new array is also simpler.

6.6 Rewriting Debugging Code

Previously, each routine `MODroutine()` (where "MOD" was the name of the routine's module and "routine" was the actual name) in ObServer had a variable declaration of the form:

```
SVR_PROC                P = MOD__routine;
```

where `MOD__routine` was `#defined` to some unique number. Additionally, each module had its own `MODrtn_proc_name()` routine that consisted of a large "switch" statement to translate `MOD__routine` into a character string containing the routine's name. These items were used for debugging purposes to print routine names when they failed. Also, each routine used `M__PRTenter`, `M__PRTexit` and `M__return` (now called `M__PRTreturn`) macros that took strings containing the routines' names as parameters and printed these names (when desired for debugging purposes) when the routines were entered or exited. The need to type in the text of each routine's name at its beginning and at each point at which the routine could exit was not only inconvenient, it also wasted a good deal of space.

So, all the `SVR_PROC` declarations were changed to ones of the form:

```
static char             *P = "MODroutine";
```

and the `M__PRT` macros above were modified to use this variable, rather than a separate macro parameter. Furthermore, all the `MODrtn_proc_name()` routines were removed since a translation was no longer necessary. This change removed 80 KB throughout the ObServer executable files.

6.7 Other Rewriting

Various portions of ObServer were poorly coded. The changes that were made include:

- Cleaning up some comments.
- Renaming some local variables that hid global ones, especially in the Binder.
- Fixing some inconsistent caller/callee type errors.
- Making non-void routines always explicitly return 0, so that they no longer cause errors when optimization is turned on (especially in the Deadlock (DEAD) module).

- Making various other minor modifications so that the compiler now produced no warnings and lint produces many fewer than it used to.

Additionally, in the "makefile", dependencies were fixed up, unnecessary files were removed and general cleaning up was performed.

6.8 Summary

The changes mentioned above not only resulted in code that was easier to use and modify, they also produced executables that were far less bulky. The final size changes produced were as follows:

"OBSbinder" — 40 KB smaller
"OBSclient" — 40 KB smaller
"OBSdaemon" — 24 KB smaller
"OBSserver" — 160 KB smaller

7 Current Performance

The overall effect of the above changes was quite significant. Table 5 compares the results of the benchmark tests before and after the changes were made. For a description of the row and column headings in Table 5, see Section 3.2 above.

Table 5: Pre- and Post-Enhancement Times

Test	*Parameter(s)*	*Before*	*After*	*Improvement*
Create Database	100 Objects	52.2	40.4	23%
	250 Objects	49.4	37.1	25%
Create & Access	No Cache, Clustered Access	575.9	259.7	55%
	250 Objects, 2 MB	607.5	282.7	54%
Average Access	No Cache, Clustered Access	0.102	0.055	46%
	250 Objects, 2 MB	0.091	0.054	41%
Average Commit	No Cache, Clustered Access	0.201	0.048	76%
	250 Objects, 2 MB	0.219	0.068	69%

The next two tables are presented to provide a final contrast between the numbers given in [FZE90] and the present. For a description of the row and column headings in Tables 6 and 7, see Section 3.1 above.

The cause of the rather large increase in file system writes in Table 6 is unclear. The most important fact to note from this table is that the overall database creation time is more than four times faster than it originally was.

The significant fact demonstrated by Table 7 is that, depending on how slow the original numbers were, object accesses are now between 13 and 44 times faster than they used to be! It is also worth noting that random requests have generally *faster* response times than clustered ones. This is quite unexpected and merits further investigation.

Table 6: Database Creation Statistics—Final Effects

Item	*Benchmark*		*Present*		*Average*
	100 *Objects*	250 *Objects*	100 *Objects*	250 *Objects*	*Improvement*
Time	177	245	41	38	81%
Minor Page Faults	413	408	327	303	21%
Major Page Faults	928	925	3	4	> 99%
File System Reads	90	65	3	7	94%
File System Writes	653	627	1072	1066	−67%

Table 7: Average Object Access Times—Final Effects

Segment Cache, Clustered Accesses

Cache Size	*Objects*	*Benchmark*	*Present*	*Improvement*
2 MB	100	0.90	0.052	94%
	250	0.80	0.056	93%
1 MB	100	1.05	0.052	95%
	250	0.90	0.053	94%

Segment Cache, Random Accesses

Cache Size	*Objects*	*Benchmark*	*Present*	*Improvement*
2 MB	100	1.25	0.050	96%
	250	0.70	0.053	92%
1 MB	100	0.95	0.051	95%
	250	1.10	0.052	95%

No Segment Cache, Clustered Accesses

Cache Size	*Objects*	*Benchmark*	*Present*	*Improvement*
2 MB	100	1.80	0.049	97%
	250	1.90	0.053	97%
1 MB	100	1.80	0.057	97%
	250	2.15	0.061	97%

No Segment Cache, Random Accesses

Cache Size	*Objects*	*Benchmark*	*Present*	*Improvement*
2 MB	100	2.10	0.049	98%
	250	2.30	0.052	98%
1 MB	100	1.45	0.057	96%
	250	2.25	0.064	97%

8 Future Areas of Research

Although the changes described above provided significant performance improvements, there are still some areas of the server that could use improvement. In addition, there are also some aspects of ObServer whose behavior is inconsistent with the documentation.

8.1 Reimplementing Secondary Storage

The server's actual database file is mainly used for storing segments. However, the front of the database file contains a number of non-segment structures that the Secondary Storage module uses to maintain the database file. These structures are, in order:

1) A Super Server Block (SECS_sblock).
2) Two Free Segment ID Bitmaps (SECS_fsidb).
3) An old Free Segment ID Bitmap (SECS_fsidb_old).
4) Two Free Extendible Hashing Leaf Page Bitmaps (SECS_flb).
5) Two Free Block Bitmaps (SECS_fbb).
6) Two Extendible Hash Tree Directories (SECS_xdir).
7) Two Sets of Extendible Hash Tree Leaves (SECS_leaf).
8) An Offset to Shadow Page Offsets (SECS_shadowoffs).

The Super Server Block keeps track of the size of database blocks, the maximum size of the database file itself, and the compactor delay (the number of seconds to wait after last SECS call before starting the compactor). The Free Segment ID Bitmap tells which segment numbers can be used. The Free Extendible Hashing Leaf Page Bitmap says which hashing leaf pages are free. The Free Block Bitmap keeps track of which blocks are free. The Extendible Hash Tree Directory and the Extendible Hash Tree Leaves handle the mapping from Segment IDs to database block numbers. Where there are two copies of an item, this is for shadowing.

Aside from maintaining control information (in item 1), the SECS module mainly uses these structures to implement two functions: 1) mapping Real SIDs to database offsets and sizes (items 4, 6 and 7), and 2) keeping track of unused Real SIDs (items 2 and 3) and database blocks (item 5). Although this scheme currently works, there are a number of reasons why it should be replaced. They are listed below in order of increasing significance:

- The code in the SECS module is difficult to read and modify. This is in large part due to the extendible hash table that is used to keep track of segment locations (the Segment Pointer Table). Manipulations of this structure are scattered throughout the module and while individual operations are well documented, the purposes as a whole are often unclear.
- The SECS module does not actually do all that it should. It is supposed to periodically compact the database, but this facility has been disabled, presumably because it corrupted the data. Because the code is currently so convoluted, it seems unlikely that anyone will be willing and able to figure out what is wrong and fix it.
- But most importantly, the scheme itself is inefficient. Accessing or updating the SPT or other data structures kept at the front of the database file is slow because none of that information is cached. While presumably some of this information could be cached, it again seems unlikely that anyone will figure out a good caching method that is both useful and assures database

consistency. Again, this is in large part because of the difficulty of working with the SECS module code itself.

Furthermore, it should be noted that database consistency is currently assured by having the SECS module toggle a shadow switch before each write to the database. This forces every changed structure (except the segments themselves) to be written to the database each time they are changed. For example, this means that currently, each call to the Secondary Storage Write routine (`SECSwrite()`) causes six calls to *lseek* and *write*, both of which are fairly time-consuming.

In light of these problems, it is probably a good idea to revamp the entire Secondary Storage module, including the information it stores. The first item to deal with is mapping Real SIDs to database offsets and sizes. As described in Step 2 in Section 4.3 above, this procedure is the second half of the process of locating a segment. However, it need not be. There is no reason why Internal UIDs could not be modified to store Real SIDs and the Segment Mapping File could not instead map these Real SIDs directly to {*Offset*, *Real Size*, *Virtual Size*} triples. Not only would this eliminate the extra level of indirection currently necessary in object lookup, it would also allow all the data necessary for this lookup to be cached in main memory. Furthermore, it would eliminate the need for storing extra information (Virtual SIDs). Both the concept of Virtual SIDs and the Segment Pointer Table with its elaborate extendible hashing scheme (implemented in items 4, 6 and 7 above) could simply be thrown out. This proposed procedure is shown graphically in Figure 2.

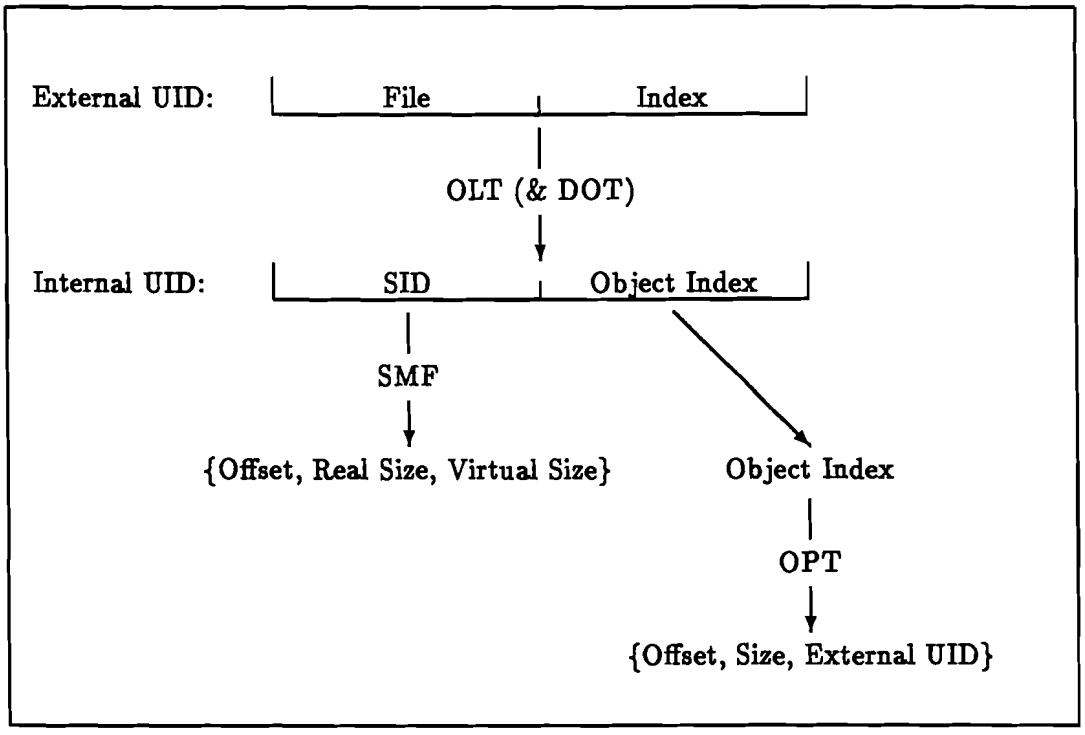
The only concern with making this change is assuring that the database could not become corrupted in the event of a server crash. It must always be possible to make a segment contain the exact same data that it did as of the last checkpoint. Currently, the use of both Virtual and Real SIDs is necessary to assure this. However, as long as segments are given new locations the first time they are written after each checkpoint, there is no need for new SIDs too. If the server crashes, the backup version of the SMF can be used to retrieve the old mappings. Otherwise, the old locations can be reused after the database has been checkpointed. It should be noted, though, that the segments' old locations would have to be stored in some temporary main memory data structure until this time. If the server crashed before the checkpoint, it is critical that the data in that memory not have been touched.

The Segment ID bitmaps (items 2 and 3 above) could be incorporated into this modified SMF module. Since SIDs are initially given out sequentially, the SMF would simply need to keep track of the highest-numbered SID used. When segments got deleted their entries in the SMF could be marked as unused and incorporated into a free list of SID entries. This would require no additional space and the the SMF module would merely need to maintain a pointer to the most recently deleted SID entry. Again, the locations that these entries pointed to would have to be stored in some temporary main memory data structure until the next checkpoint, in case the server crashed.

The Free Block Bitmap (item 5) would probably also be better placed in its own file and kept as a free list of {*Offset*, *Size*} pairs. This information could also be cached. As mentioned above, care would have to be taken not to reuse recently freed database blocks until after the next checkpoint. This storage system would have the additional advantage of not having to allocate a bitmap one 1024th of the size of the database just to indicate which blocks were and were not in use (as is currently necessary).

With the elimination of this last changeable database header structure, the Shadow Page Offsets (item 8) could also be deleted. Thus, only the Super Server Block (item 1) and the segments

Figure 2: Proposed Object Mapping Procedure



themselves would need to remain in the database file. This set of changes would yield a significant performance improvement and would almost surely be worth the time necessary to implement them.

8.2 Other Areas of Improvement

The following are additional suggestions for future work to improve the performance of ObServer. For other, less critical suggestions, as well as a list of various items to investigate and a short list of rejected ideas, see the file "suggestions".

Object Location Table

As mentioned above, the Object Location Table is currently kept in a numbered set of files. An External UID of the form {*File*, *Index*} is looked up by reading in the entry at index *Index* in OLT file *File*. This method requires no computation and only a single disk access to look up any entry. If few or none of the database objects that are ever created get deleted, this is clearly an optimal algorithm.

However, if a large percentage of the objects are transient, a like portion of space will be wasted in the OLT files with tombstones. At 8 bytes per object ever in the database, this could become very significant. Assuming, for example, that 8 GB can be used solely for OLT files, this sets a hard limit of 1 billion objects in the database ever, even if no more than a thousand are around at any given time. This is undesirable, especially in light of the fact that the choice of using 64-bit

UIDs (instead of 32-bit ones) was presumably made to allow more than 4 billion objects ever[7]. If ObServer is to reach its full potential of a seemingly infinite number of available UIDs, the OLT entries should be stored in a more flexible structure such as a *B*-tree[DEK73, pp. 473-480].

Duplicate Object Table

Currently, the Duplicate Object Table has no main memory cache structure. Since most existing ObServer applications do not replicate or migrate objects, this is not a problem. However, if this functionality is to be used in the future, a cache structure like those in the OLT or SMF modules would be very useful.

Also, the DOT, like the OLT, does not reclaim its memory. Every object that was ever in the DOT leaves a 48-byte entry behind, never to be reused, even if the object is deleted. If a number of replicated or migrated objects get deleted, this becomes an inefficient use of space. In this case, a better system would be for the DOT module to maintain a pointer to the most recently freed DOT entry and for each free entry to have a pointer to the next free one after that (e.g. a free list).

Main Memory Segment Table

An LRU algorithm is used to select which segment should be paged out from the Main memory Segment Table when space is needed. The current implementation of this algorithm involves checking the timestamp of each node in the MST. This is an O(n) operation and is unnecessary. Entries in the MST could be chained together in doubly linked lists, ordered by the most recent access (like the data structures in the OLT and SMF modules) and then the least recently used segment would always be immediately available.

File Memory Management

Currently, the responsibility for opening and closing files, as well as caching their data (if desired), lays with the individual modules that use files. As a result, caching is spotty and a number of modules have redundant file-open and -close code. Furthermore, each module only allows itself one open file at a time, which leads to unnecessary file-opens and -closes. From both a software engineering standpoint and a performance standpoint, this situation is undesirable.

A good solution to this problem would be for a COMMON group module to be written to provide a global file memory management system. It is possible that the *mmap* memory-mapping facility could be used to help implement this, though maintaining compatibility with systems that did not support *mmap* would have to be considered. Besides removing the redundant code, this could allow file descriptors to be used where needed. Furthermore, caching could be uniformly provided for all modules that do not need to write their data straight to disk for database consistency reasons[8].

[7]Note that currently the constants defined in "CMNuid.h" provide for up to 65534 ($2^{16} - 2$) OLT entries per file in up to 1001 files for a current object limit of just under 65.6 million. In theory, either the file number or entry number limits can be raised up to $2^{28} - 1$ for up to 2^{56} objects though of course this would require a quarter of a billion OLT files, each with two billion characters in them. Also, MAX_FILE_ID (currently defined at 2000 in "CMNglob_loc.h") would have to be increased. Finally, Steve Reiss' Garden[SPR87] project currently assumes that only the lower-order 16 bits of the file and index numbers are valid and he should be informed of any changes.

[8]This should only preclude the database and log files from using a cache. The MST provides a cache for the database file and the log files must be as non-volatile as possible. Otherwise, ObServer's recovery system (which

Perhaps most significantly, such a system would make checkpointing much faster. As noted in Section 4.2 above, at checkpoint time, the entire contents of each persistent server structure (except the database file itself) are dumped to backup or shadow files. This is done regardless of how much or little has actually changed since the last checkpoint. A global file memory management system could keep track of which blocks of data would need to be updated in the shadow copy and only these blocks would need to be rewritten. The use of "backup" copies—which is confusing alongside the "shadow" system—could then be eliminated.

Writing to Ports

The Server Message Module Send routine (`SMMsend()`) currently concatenates all its outgoing messages into a single buffer and passes this buffer to the Communication module's Write to Port routine (`COMMwrite_to_port()` in the COMMON group). If a new Write to Port routine were written that used *writev*, this concatenation would be unnecessary. However, maintaining compatibility with systems that did not support *writev* would have to be considered.

Reading from Ports

Currently, the Communication module's Read from Port routine (`COMMread_from_port()` in the COMMON group) only reads as much data as its calling routines request. Though this may seem logical, it actually results in an unnecessary amount of calls to the *read* routine since messages waiting at ports are often read in two pieces: first the header, then the actual data. One possible way to improve the efficiency of reading from ports (at the expense of performing some additional data copying) would involve creating buffers for each open port and reading as much as possible with each call to the Read from Port routine. If this routine were modified to use the *readv* call, it could first read the amount of data specified by its calling routines and then put the rest into the port's buffer. Then, on subsequent calls, the Read from Port routine could first look to this buffer for more data before going to the system call again.

Multigranularity Locking

Dave Langworthy suggested using multigranularity locking. "Suppose you want to read locks on all the objects in a segment and write locks on a few objects in that segment. With multigranularity locking you could get all the read locks by just read locking the entire segment and then only have to obtain the write locks. For that matter you could write lock the entire segment with just one table reference"[Personal letter]. This is probably a good idea. Currently, locks are kept on a per-uid basis in a giant hash table. Thus locking a segment with 250 objects, for example, requires 250 lock table entries. Segment-level locking would prevent this.

8.3 Unnecessary Modules

Additionally, there are two modules in the SERVER group which should probably be deleted. The first is the Segment Hashing Table (SHT) module which keeps track of open segments on a per-transaction basis. The information stored in SHT entries repeats what is in the Main memory

---

treats open files as volatile and uses log and backup files for recovery purposes) makes the immediate writing of data to files unnecessary.

Segment Table, except that the SHTs only know about the segments for one transaction, not the entire set of transactions. Therefore, if each transaction has a number of segments open and none of the other transactions use the same segments, this might be useful to avoid searching the whole MST. On the other hand, SHT entries do not show what segments other transactions have in memory and are therefore unable to suggest using these. Furthermore, SHT entries are not updated when the MST swaps segments out so its suggestions might be useless or wasteful. Rich Kogut notes: "I don't know if maintaining the SHT is worth the trouble or not. If anyone ever reactivates the segment context [Segment Group] code, it might be easy to establish an implicit segment context instead of putting the information in the SHT" [Personal letter].

The Segment Mapping Table is also not really necessary. There is no reason why Virtual to Real SID mappings cannot be written straight to the Segment Mapping File. If the server crashes, the old SMF can be restored from its ".bak" files; the real SIDs are not "lost" until the next checkpoint. The only real use of keeping track of which mappings have been created since the most recent checkpoint is to keep the Segment writing routine (`SEGwrite()`) from unnecessarily creating more Real SIDs and to know which old Real SIDs can be recycled after checkpoints. When the Segment writing routine is called, it first sees if a Virtual SID is already mapped in the SMT—if so, there is no need to create a new real location for the segment because one has already been created since the last checkpoint. Also, the routine that converts the SMT to the SMF at checkpoints (`SMTconvert_smt_to_smf()` returns a list of old mappings that were changed since the last checkpoint and this list is used to decide which Real SIDs can now be marked unused. A minor modification to the SMF to keep track of "dirty" mappings would take care of both these needs and make the SMT completely unnecessary.

Also, see Section 6 for a discussion of modules that have already been deleted but may be of interest in the future.

8.4 Inconsistencies with the Documentation

Finally, there are a few places in ObServer where the actual behavior is inconsistent with what is in the documentation. In the items below, "the documentation" refers to "ObServer II Server Interface Specification"[MFF88].

Group Mode

The following paragraphs are from the documentation:

> In ObServer II, the server executes in one of two modes : *normal* or *group*. In normal mode, any object modifications that a transaction makes are installed in the database even if the transaction aborts. Although this violates the standard notions of transaction behavior, it makes no assumptions about which transactions are privy to uncommitted object changes. In group mode, it is assumed that *all* transactions are cooperating and are thus dependent on each other's behavior. While the server executes in this mode, all transactions must abide by predefined group protocols. These protocols are described below in the server interface specifications.
>
> In a future release, it is intended that multiple levels of transaction groups will be supported and that the group protocols will be user-definable. [Section 2.3, p. 4]

Unfortunately, although this feature was researched thoroughly and discussed in [FZ89], no actual progress was made on implementing it. Furthermore, since it seems unlikely that group mode will ever be implemented, the remaining vestiges[9] of this feature should probably be removed.

Segment Groups

Similarly, the initial description of ObServer has a notion of *segment groups* which are sets of related segments[HZ87, Section 4, p. 76]. The `SVRset_sg_context()`, `SVRappend_segment_group()`, `SVRcreate_segment_group()`, `SVRfetch_segment_group()` and `SVRinquire_segment_group()` server routines are provided to work with them. However, the Segment Group Module (SGM) is not yet fully implemented and none of the above routines is actually callable. This module should either be removed completely or implemented completely.

Schedule Files

The following is also from the documentation:

> In addition to a configuration file specified by a client, each server has a unique schedule file. The schedule file determines how frequently certain server functions will be performed. The file is named DB_PATH/database_name.schedule.data. A default schedule file is found in **${OBS}/data/schedule.data**. [Section 3.3, p. 6]

The last sentence is incorrect and unclear. Currently, there is no such file, but that can easily be changed (see also the "doc_issues" file). More importantly, the sentence hints that putting a file named "schedule.data" in one's "$OBS" directory will provide default values. This is not the case although it probably should be. Such a system would allow one to have a default set of schedules, instead of having to create one for each new database. This change, if desired, would involve a simple modification to the "SCHED_read" function in "SVRsched_loc.c". In any case, a decision should be made and the documentation should reflect this decision.

Also, the documentation says: "If a keyword is omitted, the server never executes the associated function." This is currently not true. Again, minor changes to the "SVRsched_loc.c" file could make it so.

Streams

Finally, the documentation says:

> At transaction commitment, any open streams will be closed and the operations executed on the buffered data. Aborting of a transaction will result in the aborting of all open streams. [Section 9, p. 27]

This is currently false. Either the documentation or the code should be changed appropriately.

[9] These include a variable called "MODE" in the "config_file" and the `SVRexecution_mode` variable, as well as the already deleted Lock Filter module.

9 Supplementary Information

Along with the documentation provided in this paper, there are a number of supplementary files that are also of interest. These files are located in "/pro/observer/doc/masters/ghd".

"benchmarking"	—	Instructions on how to generate the benchmark data.
"changes"	—	A list of the less significant changes I made to ObServer.
"common_modules"	—	Single-sentence descriptions of each module in the COMMON group.
"deleted_files"	—	A complete list of the files that I removed from the ObServer source directory, along with reasons for each deletion.
"global_macros"	—	A description of each macro in the global module in the COMMON group.
"indent"	—	A list of formatting decisions for the **indent** program.
"overview"	—	An overview of the structure of ObServer and some pointers on coding style.
"server_modules "	—	A detailed description of each module in the SERVER group.
"suggestions"	—	Less significant suggestions, as well as a list of various items to investigate and a short list of rejected ideas.

Additionally, the "/pro/observer/doc/masters/ghd" directory contains two subdirectories. The "DATA" subdirectory contains all the data used in tables in this paper except what was taken from [FZE90]. The "MISC" subdirectory contains a few files of data and observations that were not included in this paper. This paper itself is named "FINAL.tex" and is also in the "/pro/observer/doc/masters/ghd" directory.

10 Acknowledgments

Thanks to Dave Langworthy for helping me with this throughout and to Jodi Longobardo for putting up with me throughout. And thanks to Mom and Dad for, well, everything.

A Server Modules

The following is a complete list of modules in ObServer's SERVER group:

CAT	—	Client Activation Table
CF	—	Constants File
DEAD	—	DEADlock material
DOT	—	Duplicate Object Table
LOCK	—	LOCK material
MST	—	Main memory Segment Table
NUM	—	NUMbers for objects (UIDs)
OBJC	—	OBJect Cache table
OLT	—	Object Location Table
QUE	—	QUEuing procedures
RL	—	Recovery Log
SCHED	—	SCHEDuling timer events
SECS	—	SECondary Storage
SEG	—	SEGment maintenance
SGM	—	Segment Group Module
SHT	—	Segment id Hashing Tables
SMF	—	Segment Mapping File
SMM	—	Server Message Module
SMT	—	Segment Mapping Table
SSDT	—	Server Socket Data Transfer
SVR	—	SerVeR routines
TRANS	—	TRANSaction module

For a more complete description of each module, see the file "server_modules". This includes purpose and relation to other modules, significant main and secondary memory data structures and important routines.

References

[DEK73] D. E. Knuth, The Art of Computer Programming, Volume 3: "Searching and Sorting", Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

[FZ89] M. F. Fernandez and S. B. Zdonik, "Transaction Groups: A Model for Controlling Cooperative Transactions", 3rd International Workshop On Persistent Object Systems, January 1989.

[FZE90] M. F. Fernandez, S. B. Zdonik, A. N. Ewald, "ObServer: A Storage System for Object-Oriented Applications", Computer Science Department, Brown University, CS-90-30, September 1990.

[GKM82] S. L. Graham, P. B. Kessler, M. K. McKusick, 'gprof: A Call Graph Execution Profiler', Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No. 6, pp. 120-126, June 1982.

[HZ87] M. F. Hornik and S. B. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database", ACM Transactions on Office Information Systems, Vol. 5, No. 1, pp. 70-85, January 1987.

[MFF88] M. F. Fernandez, "ObServer II Server Interface Specification", Unpublished Internal Documentation, 1988.

[SPR87] Steven P. Reiss, "Working in the Garden Environment for Conceptual Programming", IEEE Software, pp. 16-27, November 1987.

[WFT82] Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in Proceedings of the 6th International Conference on Software Engineering, IEEE, Tokyo, Sept. 1982.