BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-90-M14

Integrating Observer and Intermedia: A Case Study

by
Mala Anand

BROWN UNIVERSITY
Department of Computer Science
Masters Project

Integrating Observer and Intermedia: A Case Study

by

Mala Anand

Integrating Observer and Intermedia: A Case Study

by

Mala Anand
Brown University, 1990

Sc.M. Project

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

August, 1990

Stanley B. Zdonik
Dr. Stanley . B. Zdonik
Advisor

1. INTRODUCTION

Observer is the persistant object store and transaction manager for an object oriented database system. The goal of this project was to enhance the efficiency and performance of the system by reimplementing and performing optimizations on the existing version, such that an efficient interface could be designed to integrate Intermedia to Observer. Some of the techniques used are described below.

Some of the improvements in the system came about from, remodelling the segment interface module and structuring the segments to be dynamic, the uids of replicated objects were made unique, the priority lock queue for receiving and releasing locks was reimplemented. In the cache, the problem of insertion of a key with one hash value seemed to drastically increase the hash times, so this was eliminated by restructuring the hash table. The segment module was altered in order to enable the compaction of segments on the disk in order to minimize the external fragmentation.
There were also some code optimization techniques we applied like loop optimizations, dead code elimination and live variable analysis.

All these techniques that were used seemed to provide more functionality and efficiency for the object level server.
All these efficiency improvements, seemed to aid in the design for the integration with Observer and Intermedia. Currently Intermedia uses Ctree, a relational database, where the document objects, anchor objects, link objects and the web objects are flattened into hierarchichal tables and stored. Observer is designed to provide the tools to give any application efficient disk access. All the related observer objects are clustered together forming segments. The clustering of these objects within the segments are such , that they adapt a graph formation. Hence observer provides flexible facilities for clustering and prefetching which is useful for optimizing the performance for efficient graph traversal. It contains a transaction mechanism that allows applications to define their own policies regarding how sharing should be controlled.

The interface designed to link Intermedia with Observer gives us an opportunity to test the applicability of the storage system as a platform for hypermedia applications. Object-oriented database technology is well suited for this style of access. Hypermedia systems are based on the ability to navigate complex webs for effcient traversal. Object-oriented database technology has focused on providing the ability to do efficient graph traversal. Many people have theorized that the these two technologies would form a good match. This integration gives us an opportunity to compare the performance of intermedia with a relational backend and with an object oriented backend. This will give insight to the relative strengths and weaknesses of these storage techniques, for an object-oriented frontend application.

2. ARCHITECTURAL OVERVIEW OF THE INTEGRATION OF INTERMEDIA WITH OBSERVER

Three segments were designed to store the hypermedia information.

1) Document Segment

2) Web Anchor Segment

3) Web Link Segment

There is only one document segment in the database. There are m + n objects in this segment where m is the number of object documents and n is the number of object webs. This segment contains document identifiers for each document, and a list of segment table identifiers of web, that the document belongs. The document objects also contain their own properties, i.e. docid, docType, allRights, addTime, modTime, addPerson, docName, docpath. The document segment also contains web identifiers and their corresponding properties, i.e. docid, doctype, allRights, addTime, modTime, addPerson, modPerson, docName, docpath. It also contains a list of all the documents in the web. Each document can be in more than one web and each web can consist of any number of documents. A web itself is a special type of document.

Each document in each web, is its own segment, called a Web anchor segment. The Web anchor segment includes all the anchor objects for that document. The anchor objects contains a list of all the links connected to the anchor as well as its own properties. There are m x n web anchor segments. Each web anchor segment can have any number of objects.

The web link segment contains identifiers for each link. Each link objects has properties such as linkid, webid, linkType, addTime, modTime, addPerson, modPerson, linkExpl. It also contains a source and a destination anchor identifier for each link. There could be any number of web link segments. Each web link segment, can have any number of link objects.

The binder will manage the observer database. When the application needs to connect to a server, it first contacts the binder, which determines whether the appropriate server is already running or whether it needs to be started up. Once the binder establishes that the server is active, the client is notified, and the connection between the server and the client is complete, and the observer database is initialized.

At the time of initialization the document segment is created. When the documents and/or webs are created in the application, a corresponding entry is added to the document segment table.

When an anchor is first created in a document, the system first checks to see if a web anchor segment exists. if the web anchor segment does exist then the appropriate anchor identifiers along with their properties are inserted into the web anchor segment table. if the web anchor segment does not exist, then the segment is created and then the appropriate anchor identifiers along with their properties are inserted into the web anchor table.

When a link is created in a document, the system first checks to see if a web links egment exists. If a web link segment does exist, then the appropriate link identifiers and their corresponding properties are inserted into the web link segment table. If the web link segment table does not exist, then the segment is created after which the link identifiers and their properties are inserted into the web link segment table.

For the deletion of an anchor in adocument, the web anchor segment is scanned for the particular anchor identifier, and the coresponding entry is then removed from the web anchor segment table. Similarly, for the deletion of a link, the web link segment is scanned for the particular link identifier, and the corresponding entry is removed from the web link segment table.

3. SEGMENT MODULE INTERFACE

The interface to the segment module was remodelled in order to be able to efficiently read and write segments, given segment unique identifiers. It was reimplemented such that only new and unique segment identifiers could be explicitly requested and the old ones must be finally removed.

Segments are uniquely identified by segment identifiers. Thus there exists a segment identifier to each segment mapping. Since the size of the segments vary, the method to access them is through a pointer table. This segment pointer table was put in an extendible hashing data structure. Here the

structure we built consisted of a directory of 2 raise to the power of d words, (one for each d-bit pattern) and a set of leaf pages which contain all records with keys beginning with a specific bit pattern. The search entailed the strategy of using the leading d bits of the key to index into the directory, which consisted of pointers to the leaf pages. Then the referenced leaf page is accessed and searched for the proper record. A leaf page could be pointed to by more than one directory entry. If a leaf page contains all the records with the keys beginning with specific k bits, then it would have 2 raise to the power of (d-k) directory enteries pointing to it. This structure proved to be more efficient as the extra pointers in the directory allowed the structure to accomodate dynamic growth.
The directory, contains only pointers to pages. These are likely to be smaller than keys or records, so more directory enteries will fit on each page. We can assume that we can fit twice as many directory enteries as records on a page. When the directory spans more than one page we keep a "root node" in memory which keeps track of where the directory pages are, using the same idexing scheme. Insertion into an extendible hashing structure involves one of three operations, after the leaf page which could contain the search key is accessed. If there is room in the leaf page, the new record is simply inserted there; otherwise the leaf page is split in two (half the records are moved to a new page), If the directory has more than one entry pointing to that leaf page, then the directory enteries are split as the page is, if not then the size of the directory is doubled.

The enteries in the Segment Pointer Table are forced to be as small as possible to minimize the size of the table, however containing enough information to access segments. The minimum amount of information needed is a pointer to the segment with the coresponding segment UID. The first block would be read into main memory, and if the size is larger than a block, another disk access is neccessary to read in the remainder of the segment.

The list of free blocks is maintained in a free block bitmap with one bit per block. In order to be able to efficiently find free chunks of memory large enough to fit new segments we implemented a dynamic string searching algorithm. In the previous existing brute force approach to string searching it did not examine the effect of exploiting large memory by treating each possible M character section of the text as a key in the standard hash table. But it is not neccessary to keep the entire hash table, since only a single key is sought at a time. We need to compute the hash function for each of the possible M character sections of the text and check if it equals the hash function of the pattern. The method used for computation here is based on computing the hash function for position i in the text given its value for position i-1. The code assumes a consistent index and d=32 is used for efficiency. (As multiplcations are implemented as shifts.)

```
function  search:integer;
   const q=xxx; d=32;
   var hash1, hash2, dM, i :integer;
  begin
   dM :=1; for i:=1 to M-1 do dM:=(d*dM)mod q;
   hash1:=0; for i :=1 to M do hash1 := (hash1 * d + index(p[i])) mod q;
   hash2 :=0; for i :=1 to M do hash2 :=(hash2 * d +index (a[i])) mod q;
   i := 1;
   while (hash1 <> hash2) and (i <=N-M) do
     begin
          hash2:= (hash2 + d*q-index(a[i])*dM)mod q;
          hash2:=(hash2*d+index(a[i+M]))mod q;
        i := i + 1;
     end;
   search := i;
 end;
```

It first computes a hash value say hash1 for the pattern, then a hash value say hash2 for the first M characters of the text. It then processes the text string and computes the hash function for the M characters starting at position i for each i and comparing each new hash value to hash1. The prime q is chosen to be as large as possible but small enough that (d + 1) * q doesn't cause an overflow. This would require fewer mod operations than using the larges possible prime. This proved to be quite efficient as it took time proportional to N + M. This first fit approach used to find free chunks minimizes searches and reduces the amount of compaction that occurs. Blocks of a segment are freed upon that particular segments removal or when the segment is moved. This involves setting the appropriate bits in the free block bitmap where the contents of the block is unchanged.

4. DYNAMIC SEGMENTATION

All objects are stored in segments, therefore when an object is created and installed, the transaction indicates in which segment the object will be stored. This is achieved by calling *SVRregister_obj* or *SVRcommit_obj.* After creation, the segment identifier is not needed to update or retrieve an object. The purpose of segments is thus to physically cluster related objects. When an object is requested from the server, the server sends the entire segment to the client, in anticipation that the client will soon request other objects from the same segment. Since the segments were not dynamic, it posed a problem of the number of objects that could fit in a particular segment, in order to be able to prevent overwriting of objects. Thus the structure of the segments was redesigned to make them dynamic, such that the segment size could vary, and thus overwriting of objects was prevented. In creating this dynamic segment we assumed the combination of most varying object sizes say up to M. We performed this calculation appropriately by efficiently doing things in an appropriate order of the objects received.

```
For j := 1 to N do
  begin
     for = 1 to M  do
        if i - size [j] > = o then
            if cost [i] < [cost [i - size[j]] + val[j]) then
          begin
               cost[i] := cost [i - size[j]] + val[j];
             best[i] := j;
          end;
   end;
```

cost[i] is the highest value that can be achieved with an object size of capacity i and best[i] is the last item that was added to achieve that maximum. First we calculate the best fit we can achieve in a segment for all object sizes, when only items of one type say, A are taken, then we calculate the best we can do when only type A's and another type B's are taken. The dynamic solution, thus reduced to a

simple calculation for cost[i]. Suppose an object j, is chosen for the segment, then the best value that could be achieved for the total would be (val[j] (for the object) + cost [i - size[j]]) to fill up the rest of the segment. If this value exceeds the best value, that can be achieved without an object j, then we can update cost[i] and best[i],
otherwise they are left alone.

Thus changing it to this dynamic structure of segments prevented alot of the observer objects from being overwritten. Hence it was seen using dynamic segmentation, it implemented a dynamic structure in order to enable locality of reference, so that any number of objects that were routinely used together could be more efficiently retrieved into an applications work space at the same time.

5. SEGMENT UIDS

In the current version of observer it was found that the UID's of replicated objects are not unique, and take on already assigned UID's. This did not prove to be efficient. Hence the simplest open addressing method of "linear probing" was implemented.
If the number of objects with their UID's to be put in the hash table can be estimated in a maximum sense and enough contiguous memory is available to hold all the keys with some room to spare, then it doesn't seem efficient to use any links at all in the hash table. Thus using linear probing, when there existed a collision, (i.e. when hashed to a place in the table, which is already occupied and whose key is not the same as the search key.) then we just probe to the next position in the table, that is compare the key in the record, against the search key. There exist three possible strategies:
1) If the keys match then the search terminates.
2) If there is no record, then the search terminates unsuccessfully, otherwise probe the next position, continuing until either the search key or an empty table position is found.
3) If a record containing a search key is to be inserted following an unsuccessful search, then it can simply be put into the empty table space which terminated the search.

```
proc HASHinit;
   var i: integer;
  begin
   for i := 0 to M do a[i].key := maxint;
  end;
function Hashinsert (v: integer): integer;
    var x : integer;
   begin;
    x:= h(v);
    while a[x].key <> maxint do x := (x + 1) mod M;
    a[x].key := v;
    Hashinsert := x;
  end;
```

This method of linear probing seemed to be more efficient than separate chaining that was used. Since the table size for linear probing is greater than for separate chaining, but the total amount of memory space used is less, since no links were used. It improved performance as linear probing uses less than 5 probes, on the average, for a hash table which is less than 2/3r'd's full [1].

6. LOCKS

The lock types supported by observer are implemented with an object level granularity and provides a concurrency control scheme that can be tailored by an application to its own requirements for data consistency and resiliency. Clients use the objects by copying them from the server process into the virtual memory space of their own process via IPC prtocols. The objects are manipulated locally and saved when a client writes them back to the server IPC. A new version of the object is visible to all clients only when that version is the product of a committed transaction. The system provides a two-phase locking protocol with standard read and write locks for applications requiring serializable execution of concurrent transactions. An application may opt for the cached model of concurrency control in which objects are transferred between processes only when they have been modified by one client and are needed by another. The cached model is implemented by the addtition of two lock-types notify and writekeep.

The cached model of concurrency control, supports an environment optimized for data sharing and perfomance through the co-operative use of notify and writekeep locks. When locks are requested they get inserted into a priority queue. The structure of this priority queue, was remodelled, such that it would perform a heapsort, when all the locks on a particular object were requested. The idea was simply to build a heap structure based on the priority of the locks and then to remove and release them from the heap in order of their request. If N is the size of the entire heap and M is the number of objects to be sorted. It was implemented using M insert operations and M remove operations, putting the element removed into place variated by the shrinking heap.

```
N := 0;
for k:=1 to M do insert (a[k]);
for k:= M downto 1 do a[k]:=remove;
```

This assumes a particular representation for the priority queue. In each loop the priority queue resides in a[1..k-1]. Using this structure of heapsort for representing a priority while enabling locks improved efficiency as heapsort uses fewer than 2MlgM comparisons to sort M elements [1].

7. CACHE

The hash table that was implemented in the cache proved to be a little inefficient. The insertion of a key with one hash value drastically increased the search times for keys with other hash values. There seemed to be an easy way to eliminate this problem and improve performance, by using the technique of double hashing. The basic strategy is the same, the only difference being that instead of examining each successive entry following a collided position, we use a second hash function to get a fixed increment to use for the "probe" sequence. This was implemented by inserting u = h2(v) at the beginning of the procedure and changing x := (x + 1)mod M to x := (x + u) mod M within the while loop. This seemed to be more efficient as, double hashing uses fewer probes, on the average than linear probing[1].

The actual formula for the average number of probes made for double hashing with an "independant" double hashing function is 1/(1 - alpha) for unsuccessful search and - ln(1 - alpha)/alpha for a successful search [1]. It is more efficient as a smaller table can be used to get the same search times with double hashing as with linear probing for a given application.

8. COMPACTION

The segment module was altered in order to enable the compaction of segments on the disk in order to minimize external fragmentation. The database is compacted until contiguous chunk of memory is large enough to hold the given segment that is created. The total amount of free space that is available is kept track of, so that the WRITE call will not block while the entire database is compacted and to discover that upon completion that there does not exist enough free space in the partition.

After altering the functionality of compaction, it now migrates all the free space to the end of the disk partition. In order to be able to compact the entire database, it seemed to be more advantageous to process the segments by their offsets rather than their segment identifiers. The segments being contiguous guarantees that any used block immediately following a free one will be the first block in a segment and thus will contain its header and will contain the segment size and the segment identifier. The free block bitmap is searched for the first chunk of available segment at the end of this chunk, is moved into that free space. The first block of the segment is read into main memory, the header is examined, and if it is larger than one block, the remaining blocks are read in as well. The segment is then written out to disk updating both the free block bitmap and the segment pointer table. This protocol is repeated until the entire database has been compacted.

9. OPTIMIZATIONS

a) Dead code elimination

A variable seems to be alive at a point in a system if its value can be used, otherwise it is non-existant. Eliminating such variable's contributed to the general efficieny of the system.

b) Loop Optimizations

Three techniques were used to speed up performance.

1) Code motion: This moves the code outside the loop.

2) Induction Variable Elimination: This eliminates the induction variables from the inner loops.

3) Reduction in strength: This replaces an expensive operation with a cheaper one.

c) Local Reaching Definations:

Space for data flow information can be traded for time by saving information only at certain points and, as needed recomputing information at interevening points.

d) Live Variable Analysis:

A number of code efficiency techniques depend on information computed in the direction opposite to the flow of control in a program. In live variable analysis we wish to know say for variable x and a point p whether the value of x at p could be used along some path in the flow graph starting at p.

e) Eliminating of global common sub expressions

f) Ensuring symbolic debugging of optimized code

We ensured a way to associate an object identifier with the location it represented, such that the position of the symbol table that assigns to each variable location. A place in the global data area or in an activation record for some procedure, must be recorded by the compiler and preserved for the debugger to use.

g) Ensure scope information so references can be disambiguated to an identifier that is declared more than once.

h) Deducing values of variables in basic blocks

One solution here was to run the unoptimized version of the block along with the optimized version, to make the correct value of each variable available at all times.

The solution that was adopted to improve performance was to provide enough information about each block to the debugger. The structure used to embody the information is the dag of the basic block annotated with information about which variable holds the value corresponding to a node in the dag, at what times in both source and optimized programs.

References

[1] Sedgewick, Robert, *Algorithms,* Addison-Wesley Publishing Co., Reading, MA, 1983.

[2] Fernandez, M, Zdonik, S, Ewald, A, "*Observer: A storage system for Object oriented applications*".

[3] Skarra, A, Zdonik, S, Reiss, S, "*Observer: An object server for an object oriented database system*".

[4] Hornick, Mark, Zdonik, S, "*A shared segmented memory system for an object oriented database* ".

[5] Kogut, Richard, "*Report on implementing caching for Observer clients*".

[6] Aho, Ullman, "*Principles of Compiler Design*", Addison-Wesley Publishing Co., Reading, MA.