データベースシステム 91－5
（1992. 11. 5）

「出世魚」におけるWAKASHIの実装および評価

白 光一、寺本 圭一、天野 浩文、牧之内 顕文

九州大学工学部情報工学科

WAKASHIは「出生魚」の一番低いレベルのサブシステムで、Cプログラマーが分散共有永続データと分散共有揮発データを操作するための基本的なプリミティブを提供する。WAKASHIは、仮想メモリ方式と分散共有メモリ方式とを結合することによって、効率的且つ使い易い分散共有永続ヒープと分散共有揮発ヒープを実現している。本稿では、WAKASHIの実現法について述べる。また、WAKASHIのマルチメディアデータベース操作を評価するため、オブジェクトオペレーションベンチマークの一つ拡張を提案し、さらにこの新しいベンチマークによるWAKASHIの実行評価について述べる。

Implementation and Performance Evaluation of WAKASHI in "Shusse-Uo"

Guangyi Bai, Keiichi Teramoto, Hirofumi Amano, and Akifumi Makinouchi

Department of Computer Science and Communication Engineering
Kyushu University

WAKASHI is the most lean system in the "Shusse-Uo" project. It provides C programmers with the most primitive facilities to efficiently handle distributed shared persistent data as well as distributed shared volatile data. This paper describes implementation of WAKASHI based on the virtual-memory-based approach. This approach enables to directly map files onto a distributed shared memory. This paper also proposes an extended Object Operations benchmark so that it can be used to evaluate WAKASHI with multimedia data operations. The performance evaluation of WAKASHI using the benchmark test programs and its analysis are presented.

1 Introduction

In recent years, there are two trends in the database research. One is to develop the systems which can support advanced data-intensive applications such as CAD/CAM or CIM (Computer-Integrated Manufacturing). These advanced data-intensive applications must deal with multimedia data, distributed data, and distributed resources. These requirements are reflected in researches on new generation database management systems [TCAD90]. Along with this trend, many object-oriented persistent (or database) programming languages which can deal with complex data types are proosed and some of them are developed. Examples of such systems include O2 [DUEX90], Jasmine [AIMS90], ORION [KGBW90], Odin [HIRO91], Mandrill [YNAF91], DASDBS [SPSG90], Ontos [SOLO92], Objectivity/DB [OBJI90], ObjectStore [LLOW91], and Earth [HSK92].

The other trend is to incorporate new computer environments such as large physical memory, multiprocessors and high-speed interconnection networks to database systems[ATKI87]. Along with this trend, the new generation database management systems must be able to store and manipulate persistent data efficiently and easily for advanced data-intensive applications in such new computer environments.

We are now working on a project named "Shusse-Uo"[MAAR90]. The project is to develop a system for object-oriented and persistent (or database) programming languages in the new computer environments. The system is enhanced step by step by adding new functionalities. WAKASHI is the most lean system which provides C programmers with the most primitive facilities to deal with distributed shared persistent data as well as distributed shared volatile data.

WAKASHI trys to hide the traditional distinction between persistent data and volatile data for C programmers, and allows them to deal with distributed shared persistent data and distributed shared volatile data by the same operations. WAKASHI is based on the virtual-memory-based approach, that is, mapping or binding files into the distributed shared virtual memory. This approach is expected to be adaptable to advanced data-intensive applications, which access distributed shared persistent/volatile objects in the new computing environments. WAKASHI can be used as the storage management subsystem for DBMSs. It is actually used for INADA, which is the next upper subsystem of "Shusse-Uo".

As described in paper [BAMA92], we implemented a shared persistent heap of WAKASHI on one site, and presented its performance evaluation using the Wisconsin benchmark [BDT83] and the Object Operations version 1 (OO1) benchmark [CASK92]. However, the goal of WAKASHI is to provides distributed shared persistent/volatile heaps for multimedia data applications running on the new computing environments. So, in this paper, we discuss implementation techniques of distributed shared persistent/volatile heaps of WAKASHI, and describe an implementation using the Mach Operating System.

We want to see if WAKASHI can serve as the distributed shared persistent heaps with reasonable performance for the multimedia DBMSs. Unfortunately, there are not a benchmark that allows us to adequately measure multimedia data application performance. Although the OO1 is good for engineering database benchmark, it is not adequate for measuring the performance of WAKASHI as a distributed shared storage manager for multimedia data applications performance. It is because the OO1 benchmark does not handle large bulk data (such as image). The OO1 benchmark has to be extended, so that it can benchmark multimedia data applications. In this paper, we will describe an idea about what multimedia data applications would be, and present the performance evaluation of WAKASHI using the new benchmark.

The remainder of this paper is organized as follows: In section 2, we discuss the background of WAKASHI implementation techniques and show how to implement distributed shared persistent/volatile heaps using the Mach Operating System. In section 3, we discuss how to extend the OO1 benchmark so that it can capture the characteristic of multimedia data applications and some performance evaluation using the new benchmark test is shown. Section 4 discusses related works. In section 5, we discuss future works and remaining problems to conclude the paper.

2 WAKASHI

WAKASHI is based on the client/server architecture. As Figure 1 shows, WAKASHI Server in site2 allows client1 in site1, client2 in site2 and client3 in site3 to map (or remote) a file (or abstract memory object) into their own virtual address spaces to create distributed shared persistent (or volatile) heap.

WAKASHI Server gives clients uniform address spaces for both persistent data and volatile data, and allows clients to deal with distributed shared persistent data and distributed shared volatile data by the same operations. WAKASHI Server implements not only efficient paging between files and physical memories to support guarantees persistence of data, but also efficient paging between physical memories on multiple sites to guarantee consistency of data. Since WAKASHI Server offers transparency of file I/O operations and data distribution, it is easy to write application programs handling distributed shared persistent/volatile data.

2.1 Background of WAKASHI Implementation Techniques

This section identifies techniques and paradigms used in WAKASHI for supporting distributed shared persistent/volatile heaps on multimedia data.

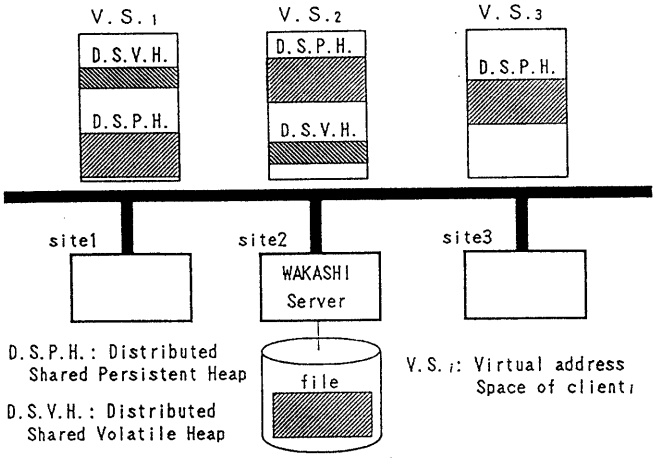

Figure 1: Distributed Shared Persistent/Volatile Heap of WAKASHI

Most of the OODBMSs are based on the clients/server architecture. All site computers are connected via a high-speed local area network (LAN) through which clients access data kept in the server. The application program retrieve objects from the server on an "object-fault" basis, when the objects is not present in the client's local object cache, the client must fetch the object from the server.

There have been two approaches for the persistent storage management for this architecture: one is based on buffer pool management, called buffer-pool-based approach (in other words two-level store), and the other is based on memory mapped file, called virtual-memory-based approach (or, single-level store).

Although many DBMSs such as O2, DASDBS, ORION-1SX, and Earth are based on the buffer-pool-based approach, but the approach has several drawbacks [BAMA92] [SHZW90]. In the approach, in-disk objects (i.e., records) and in-memory objects have different formats and are placed in different buffers. In addition to this "double buffer problem" [KGBW90], this approach have other problems such as fragmented long data problem and addressability problem (i.e., "pointer swizzling") [MAKI91][WHDE92].

The virtual-memory-based approach is expected to avoid the above mentioned drawbacks. In the virtual-memory-based approach, the database itself is mapped into the virtual address space, allowing persistent data to be accessed in the same manner as volatile data. This is in contrast to the conventional buffer-pool-based store, where access to persistent data is less direct by virtual memory address..

The idea of the virtual-memory-based approach is nothing new. A decade ago, Stonebraker [STON81] suggested, binding files into the virtual address space for DBMSs. We chose the virtual-memory-based approach for our implementation because this approach is expected to avoid the above mentioned drawbacks of the buffer pool based approach and its advantages are summarized as follows:

(1) Efficient access to persistent data. When a program accesses data in the persistent heap area, the kernel of operating system checks whether the page containing the data referenced by its virtual address is cached or not, with hardware support.

(2) Easy implementation of complex objects and long data items. Since the persistent heap area is a part of virtual address space, any component of complex object or any part of long data can be refereed to directly by its virtual address.

(3) Addressability of both persistent and volatile data. User programs and data are loaded in the same virtual address space, object identifiers become virtual memory addresses. Any data in the space is addressable and can be operated directly by the programs. It makes easy to implement encapsulation and pointer swizzling is very simple and efficient.

As far as we know, ObjectStore [LLOW91] and Cricket [SHZW90] are based on the virtual-memory-based approach. One of the different feature of WAKASHI is that WAKASHI supports distributed shared memory to implements distributed shared persistent/volatile heap, that is, mapping files into distributed shared virtual memory.

The distributed shared memory model provides processes in a system with a shared address space. Application programs can use this in the same way they use normal local memory. That is, data in the shared space is accessed through Read and Write operations. The distributed shared memory model is natural for distributed computations running on shared memory multiprocessors. Many researchers are working for constructing the distributed shared virtual memory systems [FBYR88] [NIVI91]. In a typical implementation of distributed shared virtual memory, a memory mapping routine in each processor maps the local memory onto the shared virtual address space. Memory pages are paged not only between a local physical memory and the local paging area on secondary storage, but also between physical memories of different processors.

We implemented the distributed shared persistent heap based on the distributed shared virtual memory. The key idea is to replace the local paging area on secondary storage by a user-specified file. In principle, parallel and distributed computations written for a shared memory multiprocessor can be executed on a distributed shared memory system without change, therefore, this approach can be adaptable to the new computing environments, and can be adaptable to parallel and distributed database applications.

As we have noted in the previous section, WAKASHI Server is characterized by distributed shared virtual-memory-based storage management for distributed shared persistent/volatile heap. The goal of WAKASHI is to provide C programmers with distributed shared persistent/volatile heaps on both the same site and different sites. Special virtual memory management must be made for the persistence and consistency of the distributed shared persistent/volatile heaps.

We chose the Mach Operating System [ABBG86] as the basis of our implementation because Mach provides a small number of basic abstractions and functions for execution control, inter-process communication, virtual memory management, and external memory management so that applications on Mach can be adaptable to the new computing environments such as multiprocessor workstations interconnected by high-speed networks. In particular, the Mach abstraction of memory objects made the implementation easier than it would have been under most other operating systems [YOUN89]. Mach's support for shared memory, message passing, and multiprocessors makes WAKASHI more efficient and flexible.

Mach Operating System provides EMMI (External Memory Management Interface) that allows user programs to define and manage the contents of memory object that may be mapped into virtual address spaces. Exporting this interface to the user programs simplifies the construction of complex virtual memory applications, allowing them to control sharing, consistency, and secondary storage of their data without being embedded in the operating system kernel.

A memory object is the abstraction of external memory (i.e., file on secondary storage) to be mapped on a region (in this case distributed shared persistent/volatile heap) of virtual memory in the Mach Operating System. Physical memory is used to cache the contents of memory object. To maintain and manage the memory object, a task (i.e., process in the Unix terminology), called external pager (in this case WAKASHI Server), is created. With the help of the external pager, application programs can map the memory object into their virtual address space and can access the data associated with the memory objects using virtual address.

2.2 The WAKASHI Server

The WAKASHI Server is implemented as an external pager with the EMMI. The server uses memory objects to represent abstractions of distributed shared persistent heaps for client's files, or abstractions of distributed shared volatile heaps. These memory objects are used for paging between the physical memorys and files to support data persistence, and used for paging between physical memories on different machine to support data consistency.

The WAKASHI Server is composed of the server interface and the memory objects. Clients use the server interface to create these memory objects so that clients can map memory objects into their virtual address space to create distributed shared persistent (or volatile) heaps.

The server interface includes *persistent_heap_object_create* and *volatile_heap_object_create* by which a client program request the server to create memory objects dynamically. *persistent_heap_object_destroy* and *volatile_heap_object_destroy* are for a client program to request the server to destroy the memory objects dynamically.

In order for a client to map a file on secondary storage into its own virtual address space (in other words, to allocate a distributed shared persistent heap), the client must request the WAKASHI Server to create a memory object for the distributed shared persistent heap. If the memory object is already created by other client, then the client gets a port representing the object. In this case, the memory object is shared on distributed environment. Otherwise, the server creates a new memory object for the distributed shared persistent heap. The client can uses the port to call the *vm_map* (i.e., the mapping interface of the Mach EMMI) to map the memory object into its own virtual address space to create a distributed shared persistent heap.

When a client wants to deallocate the distributed shared persistent heap, the client must request the WAKASHI Server to destroy the memory object for the distributed shared persistent heap. If the memory object is still referred to by other client, the deallocation asked by the former client is performed (i.e., its heap is deallocated), but the memory object is not destroyed. A memory object is destroyed when no client refers to it. Destruction of a memory object means deallocation of the associated port, and destruction of the associated thread.

In database applications, users sometimes create new data for initializing databases. In this case, users first create an empty file and map it into a persistent heap. Then, they insert new data in the area. At this point, it is not necessary to read data from the file to the pages into which new data is to be inserted. A function is added that allows client programs to control the page-in mechanism so that unwanted page-ins do not occur. This function is called *non_pagein_fault*. The function asks the WAKASHI Server not to execute the page-in function, even if a page-fault occurs. Another one is called *non_pageout_flush*,

which allows clients to ask the server not to page out the pages the client designates when "flush" of the persistent heap occurs.

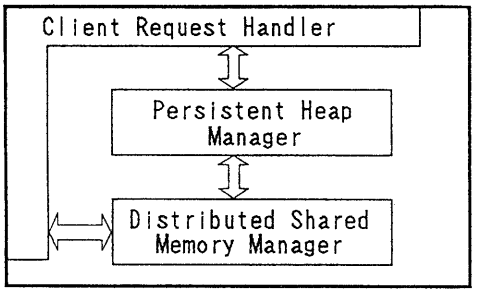

Figure 2: Memory Object's Structure of WAKASHI Server

As Figure 2 shows, the memory object of the distributed shared persistent (or volatile) heap is composed of:

(1) Client Request Handler which is the memory object's interface to clients,

(2) Persistent Heap Manager which supports persistence of data,

(3) Distributed Shared Memory Manager which guarantees consistency of data on different sites.

- Client Request Handler is the memory object's interface (i.e. the EMMI) to clients. The clients use this interface to map distributed shared persistent (or volatile) heap objects into their own virtual address space to create distributed shared persistent/volatile heap. The interface also handles all page accesses by the local or remote clients and ask either Persistent Heap Manager or Distributed Shared Memory Manager to guarantee data persistence and consistency. For example, when a page in one persistent heap is first accessed by a client, the Mach kernel uses the interface and sends a *memory_object_data_request* message to the memory object port corresponding to the heap. The memory object retrieves the data page from mapped file or a copy of other remote client, and returns that data uses the interface using *memory_object_data_provided* message to the Mach kernel.

- Persistent Heap Manager implements input/output of pages from/to a mapped file for the request from Client Request Handler. For example, when the Mach kernel sends a *memory_object_data_write* message using Client Request Handler to the memory object, the Persistent Heap Manager will output the data page to the mapped file.

- Distributed Shared Memory Manager uses a simple state machine to implement a single-writer or multiple-reader coherency protocol to keeps data consistency of distributed shared persistent/volatile heap, much like in NetMemoryServer of Mach [FBYR88].

The Mach kernel guarantees consistency among all mappings of a memory object on one site. Distributed Shared Memory Manager ensures consistency among different sites by permitting only one writable copy or any number of read only copies of any given page. Distributed Shared Memory Manager employs a state machine to trace access (i.e., read or write) and locations for each page of its memory objects. Messages from the Mach kernel trigger state transitions. When a write request is issued while readers are present, Distributed Shared Memory Manager sends a *memory_object_lock_request* through Client Request Handler to each reader to flush its copy of the page, and enters a write state until these lock requests complete. Similarly, to process a read request when a writer is present, Distributed Shared Memory Manager issues a request to clean and remove write accesses from the page and enters a wait state until the data is returned. Lock completion and data write massages remove pages from wait states, returning them to simple reader or writer states, and allowing blocked requests to be satisfied.

Distributed Shared Memory Manager uses a fairly simple page scheduling policy to ensure that progress is made when more than one request claims for a given page. The underling processor scheduling and network delays limit the rate at which pages can be reclaimed. These limitations effectively eliminate thrashing.

Distributed Shared Memory Manager also handles multiple page sizes and different data representations in order to support sharing in heterogeneous distributed systems. To support multiple page sizes, Distributed Shared Memory Manager may provide data or make lock requests in larger units than the kernel's page sizes. To support different data formats, Distributed Shared Memory Manager allows its clients to associate data types (e.g., integer or floating point) with its memory objects. When data is transferred between machines, the server may use the network message server to perform the appropriate data translations or may do those translations itself.

3 Performance Evaluation

Our previous work [BAMA92] presented the performance evaluations of a local shared persistent heap of WAKASHI using the Wisconsin benchmark [BDT83] and the Object Operations version 1 (OO1) benchmark [CASK92]. However, the work was partial because WAKASHI did not support distributed environment and the performance evaluations did not contain multimedia data or large bulk data. As mentioned earlier, the goal of WAKASHI is to provide distributed shared persistent/volatile heaps for multimedia data applications on the new computing environments. we must see if the WAKASHI Sever can manage the distributed shared persistent heaps with reasonable performance for multimedia DBMSs.

In this section, we propose a simple benchmark test for multimedia data application, and present the performance evaluation of WAKASHI using the benchmark.

3.1 Extending OO1 to Benchmark Multimedia Database Systems

It is very difficult to measure multimedia DBMS performance in a general way, since every multimedia data application has different requirements, and its way to process multimedia data (i.e., image data) is very complex. However, the multimedia data applications are quite similar at the multimedia data access level. In the access level, multimedia data are usually represented by complex objects and long objects.

At object access level, the Object Operations version 1 (OO1) benchmark presents a standard for benchmarking object-oriented database systems. The benchmark provides some basis for a comparative evaluation of engineering applications such as Computer-Aided Software Engineering and Computer-Aided Design. The benchmark database is independent of the data model provided by DBMSs, and is regarded as an abstract definition of the information to be stored, possibly as objects of single type with list-valued fields. The benchmark measures response time of target system run by a single user. The test operation set includes inserting objects, looking up objects, and navigation.

However, it is clear that the OO1 benchmark does not adequately measure multimedia database systems' performance, because it employs a small size object type (i.e., part) only. Persistent long objects are very likely to be involved in multimedia data applications and they may occupy a very large portion of persistent storage. So we add a long object type to the OO1 benchmark. The long objects represents image data which tend to be very long and are very popular in most multimedia applications.

3.1.1 Extending OO1 Benchmark Database

We add *Image* object type to the OO1 benchmark database, and define the database as consisting of two types of objects:

```
Part: {
      id:    INT,
      type:  STRING[10],
      x,y:   INT,
      build: DATE,
      to:    LIST OF {p: Part,
               type: STRING[10], length: INT}
      from:  LIST OF Part,
      refer: Image}

Image: {
      id:   INT,
      data: Bitmap}
```

Parts have unique ids from 1 through 20,000 and each part refers to three other (random selected) parts, just like parts database of OO1 benchmark. But, in our database, 1% parts also refers to a (random selected) image.

Image data has also unique id's from 1 through 20 and each image includes an id and a bitmap data. We assume that the database system to be measured allows data fields with the scalar types of INT, $STRING[N]$, $DATE$, $LIST$, and $Bitmap$.

Generally, images represented by arrays of values at the most basic level. When we store an image, we are storing a 2D array of values whose each value represents the data associted with a pixel in the image. For a bitmap, this value is a binary digit. For a color image, the value may be a collection of three numbers representing the intensities of red, green, and blue components of the color at that pixel. For simplicity, we consider black-and-white bitmap images only, and define $Bitmap$ type that presents by [512, 512] array of values.

We assume that the database itself resides on a server machine in a network, and the database server runs on this machine. We also assume that a client run on the server machine to benchmark local data access, and a client run on a client machine on the network to benchmark remote data access.

3.1.2 Extending OO1 Benchmark Measures

We dropped the *Insert* test from the OO1 benchmark, and add the *Update* one. We also allow multiuser access to database from different sites.

The following three measured.

(1) *Lookup*. Just as the OO1 benchmark, 1000 random part ids are generated and the corresponding parts from the database are fetched. When a part is fetched, for each part, a null procedure is called with the x and y position and the type of the part. If the part refers to an image data (i.e., pointer is not null), an image processing procedure is also called.

(2) *Traversal*. This is quite similar to the traversal measurement of the OO1 benchmark. That is, it is to find all parts connected directly or indirectly to a randomly selected part, up to 7 depth. whenever a part is fetched, a null procedure is called with the value of the x and y fields, and the part type.

(3) *Update*. Generate 100 random part ids and update the x, y, and *build* fields of the corresponding parts in the database.

For the *Lookup* test above, a image processing procedure is called to measure a long object access. This image processing procedure sequentially scans the bitmap data in the persistent storage. Though this seems too simple to simulate image processing procedures, we think that sequential scanning is one of the basic access patterns of image processing.

Concurrent access is also important for DBMS to allow multiuser access to the database. To see if the WAKASHI Server manage the consistency of the persistent data on the distributed computing environments, we must allow concurrent accesses by multiple users on different sites. Therefore, for the *Update* operation, a client program is initiated

for update on the server machine and another client program is initiated for lookup on the client machine. That is, the database is concurrently accessed by a local writer and a remote reader in the network.

3.2 Results of Experiments

We developed simple simulation programs in INADA [AATM92]. INADA is one of the subsystems in "Shusse-Uo", and enhances C++ with the facilities for inquiring persistent C++ objects. INADA uses the WAKASHI Server to support persistent object, object consistency, pointer swizzling and fault objects into main memory. INADA supports objects, classes, inheritance, persistent storage, and distributed sharing of objects.

We created a benchmark database in the server workstation. We did not use any index on *Part* objects or *Image* objects, in our simulation. The database comprises approximately 5 Mbytes of parts, 5 Mbytes of images, and 1 Mbytes for space overhead. As such, the size of database is approximately 11 Mbytes.

For our benchmark, we used the following system for the server where the database resides:

```
OMRON LUNA-88K with 3 M88100CPU (25MHz)
                and 32 Mbytes physical memory
OMRON SX-9100 disk controller
Mach 2.5 Operating System
```

The client machine, where we ran the remote benchmark programs, was a OMRON LUNA-88K that has 4 M88100CPU (25Mhz) and 32 Mbytes physical memory, This system also ran Mach 2.5 Operating System. Both the client and the server machines were reserved exclusively for benchmarking during the test runs.

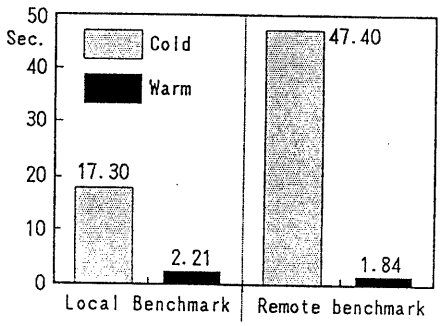

Figure 3: Results of Lookup Operation

Figure 3 shows the execution time of the Lookup measurement (it includes 10 image processing). In the "Cold" cases, all pages in the database file are paged-in once and only once during the test. In the "Warm" cases, the database can be cached on the physical memory, and the performance is very good. Figure 3 shows the most important result that WAKASHI can efficiently access the persistent long objects. In "Warm" cases, the performance is as good as the one for non-persistent long objects.

Figure 3 also shows a comparison between local database accesses and remote database accesses. In "Cold" cases and for remote database accesses, the WAKASHI Server not only executed the input data from the mapped file on the server machine, but also transfers the data to client machine using Remote Procedure Call (RPC). However, in "Warm" cases, only WAKASHI Server runs on the server machine and only one client program runs on the client machine. Therefore, no paging occurs and remote access is more efficient than local access.

Traversal measure involves finegrained 3280 parts (with possible duplicates) connected to a randomly selected part. This means that a single test causes 3280 pointer swizzlings. To measure the overhead of pointer swizzling, we simulated traversal measure in virtual memory and comparison with pointer swizzling in "Warm" cases. In virtual memory, an object pointer is a virtual memory address, and no pointer swizzling is necessary.

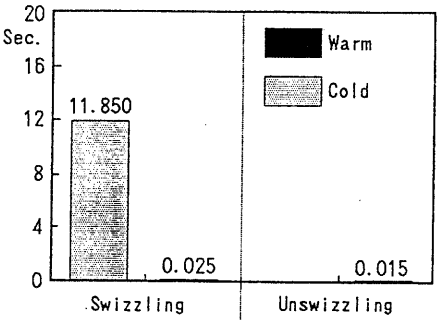

Figure 4: Comparison between swizzling and unswizzling by Traversal

Figure 4 shows the comparison result between the unswizzled case and the swizzed case. The pointer swizzling in INADA is to allocate virtual memory addresses for pages containing persistent data one step ahead of the traversal program's actual usage of the pages. When the traversal program first attempts to access a page, a virtual memory page fault occurs. This fault is intercepted by the underling WAKASHI Server which then loads the page into its preassigned location in memory. Therefore, the pointer swizzling is that the traversal program only see regular virtual memory pointers, allowing accesses to persistent objects to occur at memory speeds. As Figure 4 shows, pointer swizzling is very efficient in INADA.

Figure 4 also shows the "Cold" cases of Traversal. In this cases, some page in the database file are paged-in once during the measure.

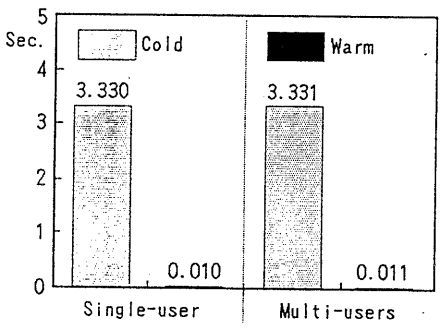

Figure 5: Comparison between single-user and multi-users by Update

To measure the overhead of locking by concurrent accesses, we measured that update operation by single user and multiple users, and compared these two measurements. In the cases of single user test, a single client program for update runs on the server machine, and no concurrent access occurs. In the cases of multiple users, in addition to a client program for update runs on the server machine, a lookup client program runs on the client machine during the simulation. That is, the database concurrent access by a local writer and a remote reader.

Figure 5 compares these two cases. In the case of multiple users, the persistent object coherence is supported by Distributed Shared Memory Manager of WAKASHI. The WAKASHI Server provides a low-level and efficient locking mechanism. As Figure 5 shows, the multiple users can access distributed shared persistent objects as efficiently as a single user.

Although these benchmark tests mentioned above are very simple, we can conclude from them that the WAKASHI Server allows

(1) Efficient access to large persistent data,
(2) Efficient pointer swizzling, and
(3) Efficient persistent data coherency.

We believe that the WAKASHI Server can be used for the storage management for multimedia OODBMSs or other data intensive applications in the distributed computing environments.

4 Related Work

The work most closely related to ours is Cricket [SHZW90]. Cricket is a database storage system, which uses the memory management primitives of Mach Operating System (i.e., EMMI) to provide the abstraction of shared, transactional, single-level (i.e., virtual-memory-based) store. Cricket also provides transparent, two-phase, page-level locking. WAKASHI is different form in that WAKASHI based on distributed shared memory approach to provide distributed shared persistent/volatile heap. Cricket does not support the distributed shared single-level store.

ObjectStore [LLOW91] is another related work. ObjectStore is an object-oriented database management system whose object storage manager is based on the virtual-memory-based approach. The ObjectStore server provides the long term repository for persistent data. A major difference between WAKASHI and ObjectStore resides in platform their implementation. ObjectStore's databases are stored in files and they are mapped into the virtual memory by using the Unix operating system's mapped-file system (i.e., *mmap*() call). The main difference is that ObjectStore uses the data-passing model to implements cache consistency, since the *mmap*() call do not supports distributed shared memory mapped files.

Camelot Distributed Transaction System [SPEC87] uses the EMMI of Mach to provide a persistent store based on the virtual-memory-based approach. In contrast to WAKASHI, the Camelot's persistent store is not directly accessed by client programs. Instead, it is accessed only by a "data server" that manage all the persistent data.

Napier88 [KIDE90] also takes a similar approach and also uses the Mach Operating System's external pager to perform paging for persistent data. This implementation technique is very similar to ours. However, every data in Napier88 is persistent, while both volatile and persistent heaps are supported by WAKASHI so that the C++ application programs can equally operate on both distributed shared persistent and volatile data.

5 Summary and Future Plans

WAKASHI is constructed on the Mach Operating System. Using WAKASHI the traditional distinction between persistent data and volatile data is hider for C programmers, so that users can deal with distributed shared persistent data and distributed shared volatile data in a very similar way.

one of important characteristics of WAKASHI is mapping or binding of files into the distributed shared virtual memory. This approach is expected to be adaptable to advanced data-intensive applications, to provide high-performance access to distributed shared persistent objects, and to be adaptable to new computing environments such as large main memory, multiprocessors, and high-speed network.

In order to prove that some these expectations are true, we also proposed new benchmark tests for multimedia data applications, and showed its result. The result suggests that the WAKASHI Server is fairly good in its performance and may be used by database systems or other data intensive applications that handle multimedia data in the distributed environments.

WAKASHI is a centralized server to manage distributed shared persistent/volatile heaps. Such a centralized solution has several drawbacks [FBYR88], since the server may become a bottleneck, proximity of a client to the server may affect its performance and the server host performs an unfair amount of computation, possibly degrading other tasks on that host. We are now developing the distributed version of WAKASHI that may avoid the above mentioned drawbacks.

In addition we have three other works to be done, as follows.

Transaction support by a distributed shared persistent heap consists of interconnected local caches and memories observing some protocol. We must consider a combined method from the following research areas: distributed database management systems due to transaction support; operating systems due to primitives required for memory interface and handling of association between transactions and processes.

The Mach kernel chooses pages to be replaced using an approximate LRU (Least Recently Used) algorithm. However, some database operations have access patterns that may make the LRU inappropriate [STON81]. By an extending the Mach EMMI, a user-level page replacement by external pager may be implemented. For each persistent heap area, we may provide specific non-LRU page replacement strategy which better fits to the area.

For any OODBMS to access objects, it calls the methods associated with them. For calling the methods, the OODBMS needs dynamic linkage which allows compiled functions to be loaded in main memory. Every time the OODBMS needs to access an object, it dynamically loads the binary file containing the access functions, if they are not already loaded [AIMS90]. The virtual-memory-based approach for storage management allows users to map not only a data file, but also a program (method) file into its own virtual address space. WAKASHI will allow users to map its objects into its own virtual address space, so that data and methods will be uniformly managed.

REFERENS

[AATM92] H. Amano, M. Aritsugi, K. Teramoto, and A. Makinouchi, "Designing C++ Primitives for the Persistent Programming Language INADA", Proc. 44th IPSJ conf., 2H-3, March 1992 [in Japanese].

[ABBG86] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A.Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development", USENIX, July 1986, pp.93-112.

[ABDD90] M. Atkinson, F. Bancilhon, D. DeWitt, L. Dittrch, D. Maier, and S.Zdonik, "The Object-Oriented Database System Manifesto", The Committee for Advanced DBMS Function, Memorandom No. UCB/ERL M90/28, April 1990.

[AIMS90] M. Aoshima, Y. Izumida, A. Makinouchi, F. Suzuki, and Y. Yamane, "The C-based Database Programming Language Jasmine/C", Proc. 16th Intel. Cong. on VLDB, August 1990, pp.539-551.

[ATKI87] Malcolm P.Atkinson, "Types and Persistence in Database Programming Languages ", ACM Computing Surveys, Vol.19, No.2, June 1987, pp.105-190.

[BAMA92] G. Bai and A. Makinouchi, "Implementation and Evaluation of a New Approach to Storage Management for Persistent Data — Towards Virtual-Memory Databases", Proc. of the Second Far-East Workshop on Future Database Systems, Kyoto Japan, April 1992, pp.211-220.

[BDT83] Bitton, D., DeWitt,D.J., and Turbyfill,C., "Benchmarking Database Systems: A systematic Approach", 1983 Proc. of VLDB Conference, Oct., 1983, pp.8-19.

[CASK 92] R. G. G. Cattell and J. Skeen, "Object Operations Benchmark", ACM Transactions on Database Systems, Vol. 17, No. 1, March 1992, pp.1-31.

[DEUX90] O. Deux et al. "The Story of O2", IEEE, Transactions on Knowledge and Data Engineering, Vol.2, No.1, Mach 1990, pp.91-108.

[FBYR88] A. Forin, J. Barrera, M. Young, and R. Rashid, "Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for MACH", CMU-CS-88-165, August 1988.

[HSK92] H. Hayata, N. Sato, and M. Kobe, "Design and Implementation of Core in OODBMS Earth", Conf. Database Systems, July 1992, pp.59-68.

[KGBW90] W. Kim, J. F. Garza, N. Ballou, and D. Woelk, "Architecture of the ORION Next-Generation Database System", IEEE, Transactions on Knowledge and Data Engineering, Vol.2, No.1, March 1990, pp.109-124.

[KIDE90] G. Kirby and A. Dearle, "An adaptive browser for Napier88", University of Standrews Research Report CS/90/16, 1990.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System", Communications of the ACM, Vol.34, No.10, October 1991, pp.50-63.

[LIHU89] K. Li, and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", ACM Transactions on Computer Systems, Vol 7, No. 4, November 1989, pp.321-259.

[MAAR91] A. Makinouchi and M. Aritsugi, "The Object-Oriented Persistent Programming Languages for Multimedia Databases", Technical Report CSCE-91-C04, March 1991.

[MAKI91] A. Makinouchi, "Architectures of the Object-Oriented Database Management Systems", Information Processing, Vol. 32, No. 5, 1991, pp.514-522 [in Japanese].

[MCAR90] D. McNamee, and K. Armstrong, "Extending The Mach External Pager Interface To Accommodate User-Level Page Replacement Policies", Mach Workshop, 1990, pp.17-29.

[NIVI91] B. Nitzberg and V. Lo, "Distributed Shared Memory: A survey of Issues and Algorithms", Compute, Vol. , No. , August 1991, pp.52-60.

[OBJI90] Objectivity, Inc., Objectivity/DB, 1990.

[RICA89] Richardson, E. J. and M. J. Carey, "Persistence in the E Language: Issues and Implementation", Software-Practice and Experience, Vol.19, N0.12, December 1989, pp.1115-1150.

[SPSG90] H. J. Schek, H. B Paul, M. H. Scholl, and G. Weikum, "The DASDBS Project: Objectives, Experiences, and Future Prospects", IEEE, Transactions on Knowledge and data Engineering, Vol.2, No.1, March 1990, pp.25-43.

[SHZW90] Shekita, E. and Zwilling, M., "Cricket: A Mapped, Persistent Object Store", Proc. the fourth International Workshop on Persistent Object Systems (1990), Morgan Kaumann. pp.89-102.

[SOLO92] Soloviev, V., "An Overview of Three Commercial Object-Oriented Database Management Systems: ONTOS, ObjectStore, and O2", SIGMOD RECORD, Vol.21, No.1 (1992), pp.93-104.

[SPEC87] A. Z. Spector, "Distributed Transaction Processing and The Camelot System ", CMU-CS-87-100, January 1987.

[STON81] M. Stonebraker, "Operating System Support for Database management", Communications of the ACM, Vol.24, No.7 July 1981, pp.412-418.

[TCAD90] The Committee for Advanced DBMS Function, "Third-Generation Database System Manifesto", Memorandom No. UCB/ERL M90/28, April 1990.

[VSK90] F. Vaughan, T. Schunke, and B. Koch, "A Persistent Distributed Architecture Supported by the Mach Operating System", Mach Workshop 1990, pp.123-140.

[WARO91] Y. Wang and L. A. Rowe, "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", Proc. ACM-SIGMOD pp.367-376.

[WIDE92] S. J. White and D. J. Dewitt, " A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies", Proc. of the 18th VLDB Conference, Canada, 1992, pp.419-431.

[YNAF91] Y. Yamamoto, M. Namiko, M. Asami, K. Furukawa, and K. Sato, Masamichi Kato, and Takeo Maruyama, "An Object-Oriented Database System: Mandrill — its Overview— ", Proc. of 43th IPSJ conf., October 1991 [in Japanese].

[YOUN89] M. W. Young, "Exporting a User Interface to Memory Management from a Communication-Oriented Operating System", CMU-CS-89-202, November 1989.