Dalí: A High Performance Main Memory Storage Manager

H. V. Jagadish **Daniel Lieuwen** **Rajeev Rastogi** **Avi Silberschatz**
S. Sudarshan

AT&T Bell Labs.
600 Mountain Ave., Murray Hill, NJ 07974
{jag,lieuwen,rastogi,silber,sudarsha}@research.att.com

Abstract

Performance needs of many database applications dictate that the entire database be stored in main memory. The Dalí system is a main memory storage manager designed to provide the persistence, availability and safety guarantees one typically expects from a disk-resident database, while at the same time providing very high performance by virtue of being tuned to support in-memory data. Dalí follows the philosophy of treating all data, including system data, uniformly as database files that can be memory mapped and directly accessed/updated by user processes. Direct access provides high performance; slower, but more secure, access is also provided through the use of a server process. Various features of Dalí can be tailored to the needs of an application to achieve high performance — for example, concurrency control and logging can be turned off if not desired, which enables Dalí to efficiently support applications that require non-persistent memory resident data to be shared by multiple processes. Both object-oriented and relational databases can be implemented on top of Dalí.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

1 Introduction

There are a number of database applications, particularly in the telecommunications industry, where very high performance access to data is required. Such applications typically require high transaction rates, coupled with very low latency for transactions, and stringent durability and availability requirements. As as example, consider a real phone-company application where phone call data is recorded, and queries against the data can be issued. The application requires several thousand (albeit small) requests (lookups/updates) to be processed per second, with less than 50 milliseconds latency for lookups, and less than a few minutes of downtime a year. Such applications have been previously implemented as stand alone programs that run in main memory and provide their own (usually limited) forms of sharing and persistence mechanisms. It is increasingly being realized that the core requirements of these type of applications would best be met by using an underlying main-memory storage manager that supports full functionality such as transaction management, concurrency control and recovery services. Using the same storage manager across multiple applications can greatly reduce development costs.

The increasing availability of large and relatively cheap memory also suggests that more database applications could reside entirely or almost entirely in main memory. Such applications will experience performance benefits by having data cached in main memory. However, if the storage manager supporting such applications is tailored to main memory, significant additional performance benefits can be achieved, as shown in [LSC92]. Thus, storage managers tailored to main memory would also be ideally suited for such databases.

The Dalí[1] system, implemented at AT&T Bell Laboratories, is a storage manager for persistent data that has been optimized for environments in which the database is main-memory resident. Dalí uses a memory-mapped architecture, where the database is mapped into the virtual address space of the process. While Dalí can be used in systems where the database is larger than main-memory, the architecture of Dalí, in particular its recovery mechanism, has been designed to deliver high performance when the database fits into main memory. Parts of the Dalí architecture that differ from storage manager architectures for disk-based systems include the following:

- The use of the recovery mechanism of [JSS93], which is tailored to high-performance main-memory databases.
- The partition of the database into *database files* that can be memory mapped into *shared memory* by user processes. The benefits of this are:
  - Data can be accessed directly by user processes, avoiding costly interprocess communication.
  - User processes can memory map selected database files rather than the whole database, enabling users to map into memory only the parts of the database relevant to the application.
  - System data, such as log records and lock structures, are themselves stored in a *system database file*, and can be memory mapped and manipulated directly by user processes.
  - Multiple processes can concurrently access the database.
- Tolerance to software faults. Fault-tolerance features are particularly important in the context of Dalí since processes may access and update shared data directly for performance reasons. The fault-tolerance features provided by Dalí include recovery from processes that fail while updating shared data, and detection and recovery from corruption of shared data.
- Tunability of the system, which enables tailoring of the database to achieve high performance. Dalí allows various features, such as concurrency control and logging, to be turned off if not required. Users can also control the partition of Dalí system code between libraries linked with the user process, with direct access to the database, and code executed at server processes.
- Support for compression of individual data 'items'. Data items are structured as a fixed size header and a variable sized data component, which facilitates data compression.

Dalí provides many other useful features. For instance, high availability is achieved through support for remote backups and hot spares and support for online schema and software changes.

By providing direct access to data, Dalí is well suited for a "client as server" environment [CD+94], where users add software on top of the storage manager to build a server with a higher level of functionality. Dalí also provides an interface to access data from remote sites, based on remote procedure calls. Coupled with the ability to split the database into files, this feature can be used to implement a network of Dalí systems in case the size of the database is larger than the maximum physical memory availability on a machine.

The architecture of the Dalí storage manager is not tailored specifically to a particular data model. Thus, it is possible to build relational as well as object-oriented database systems on top of Dalí.

A prototype of Dalí is operational, and incorporates most of the features described in this paper. A prototype of MM-Ode [GLS94], a main-memory version of the Ode object-oriented database system [AG89], has been built on top of the Dalí. We are in the process of implementing a relational interface as well, on top of Dalí.

2 Related Work

A storage manager provides the core functionality of a database system, such as concurrency control, recovery mechanisms, storage allocation and deallocation, and transaction management. There have been numerous implementations of storage managers for disk resident data. These include the storage managers of Exodus [CDRS89], Starburst [HCL+90], ObjectStore [LLOW91], EOS [BP93], Texas [SSP92], the Cricket [SZ90], and QuickStore [WD94].

Independently, there has been much work in the area of main-memory databases. Much of this work has concentrated on recovery schemes and on indexing and query evaluation issues; Garcia-Molina and Salem [GMS92] provide an excellent overview of research on main-memory databases. Lehman et al. [LSC92] and Gottemukkala and Lehman [GL92] discuss the relative costs of operations such as locking and latching in the main-memory storage component of the Starburst extensible database system. They demonstrate that once the I/O bottlenecks of paging data into and out of the database are removed, other factors such as latching and locking dominate the cost of database access, and

[1] Named in honor of Salvadore Dalí, for his famous painting, "The Persistence of Memory".

they provide techniques for reducing such costs. Thus, they provide an excellent motivation for closely examining the system design of a main-memory database and tuning it to remove bottlenecks, and have thereby influenced our work significantly.

With the exception of the Starburst main memory storage component [LSC92] we are not aware of any storage manager that is tailored for main memory resident data.[2] The Starburst main memory storage component is a relational storage manager used as a component of the Starburst database system. Its emphasis is on data allocation and structuring issues, and (as of when [LSC92] was published) the main memory storage component uses the recovery manager of Starburst rather than implementing its own recovery manager. In contrast, the recovery mechanisms of Dalí are based on a recovery algorithm tailored to main memory, proposed in [JSS93]. We point out additional differences later in the paper.

Unlike Dalí and the Starburst main memory storage component, the other (existing or proposed) storage managers we are aware of are not tailored for memory resident data. The storage managers for disk-resident data can be divided into two groups. The first group consists of traditional storage managers, such as Exodus and EOS, that provide their own buffer management facilities. The second category consists of storage managers that map the database into virtual memory. Included in this category are the storage manager of ObjectStore, the Texas system, Cricket, and QuickStore.

Storage managers in this second category are more closely related to Dalí, since Dalí also uses a memory mapped architecture. However, the architecture of existing memory mapped storage managers, in particular their recovery mechanism, does not take advantage of the database being resident in main memory. For instance, ObjectStore uses pagewise checkpointing, and Texas uses a shadow paging architectures which, while providing support for old versions of data, results in slow commit processing. Also, the storage managers were designed for CAD environments where transactions are long, concurrency control at the level of pages is sufficient, and fast sharing of data is not a primary concern.

Dalí, on the other hand, is designed for high performance applications similar to traditional transaction processing applications but with much lower latency and higher throughput requirements. In a typical Dalí application, transactions are small, multiple processes may access shared data and high concurrency, especially on index structures, is important. As a result, Dalí supports item level locking. Also, the recovery algorithm used in Dalí is designed to work well with small transactions.

[2] System M [SGM90] is a transaction processing test-bed for memory resident data, but is not a full feature storage manager.

3 The Dalí System Architecture

A *Dalí storage management system* (or *Dalí system*, for short) consists of a set of "database files" along with one or more server processes. A Dalí system runs in a shared virtual memory that guarantees sequential consistency[3] [Lam79] for reads and writes. Server as well as user processes with access to the shared virtual memory can map data into their virtual memory address space; other processes can access data only via server processes that have access to the shared virtual memory.

3.1 Database Files

In a file system, a file is typically used to store related data, and users keep unrelated data in distinct files. Correspondingly, we believe it is natural, even in a database context, to store related data in a single unit, which we call a *database file*, and unrelated data in distinct database files. A user process can access and update the data in a database file by mapping the file into its virtual address space using the mmap call provided by most standard Unix based systems. Further, in such systems it is possible to ensure that the data is in fact memory resident (provided it fits in memory) using an mlock call.[4]

There are several benefits from viewing a database as a set of database files. Typically an application is not interested in all the data in a database. The entire database may be much bigger than memory, but the part of the database required by an application may fit into memory. If a database presents a flat address space and related data is scattered over the address space, it is hard to locate what data should be kept in memory for an application. Database files thus help organize the data in a database.

A database file can be memory mapped by multiple processes in the Dalí system, provided they have access to the same shared virtual memory, with sequential consistency guarantees. On uni-processor and shared-memory multiprocessor machines, the Unix mmap function call guarantees sequential consistency. Thus, a database file can be accessed directly only by user or server processes on one (shared-memory) machine. Processes running on other machines can access it only via a server process running on the local machine.

A database file consists of several partitions, each with its own protection mode (see Figure 1). The head

[3] Sequential consistency requires that operations (reads / writes) to any memory location can be serialized, and the operations from a single processor are serialized in the order in which

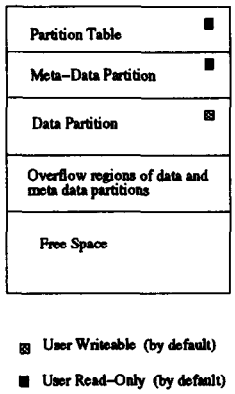

Figure 1: Database File Structure

of the database file contains a partition table that contains partition-related information and is in a partition of its own. Partitions need not be contiguous, to allow room for expansion if a partition runs out of space.

Memory allocation information is stored in a 'meta-data partition', separate from the data itself. Corruption of unprotected allocation information by stray writes has been observed to occur in object-oriented database environments, leading to severe loss of data. Hence, the meta-data partition can be write protected for all but server processes and trusted user processes.

Dalí currently stores database files in the host file system. This provides a smooth interface between the Dalí storage manager and the host file system, and operating system facilities can be used for some database functionalities such as cold backups.

3.2 System Database

All data related to database support, such as log data and lock data, is stored as a database file, which we call the *system database file*. Uniformity of data access is thereby provided. The scheme is inspired by the Plan 9 [PPTT91] philosophy of treating all data uniformly as files. The system state is thus contained entirely in the database files in the system.

As a result of storing system data in database files, user processes that run in the same shared memory as the Dalí system can access and share the data in the system database (e.g., lock information) directly, and use library routines that provide storage allocation services to allocate locks and log data. Therefore, apart from server processes to support remote accesses, the only server process *required* in a Dalí system is a system server process that performs initialization and recovery services. Storage manager services may be accessed through a server process for protection reasons,

they are generated at the processor.

[4] Super user permissions are usually required to mlock a file in memory.

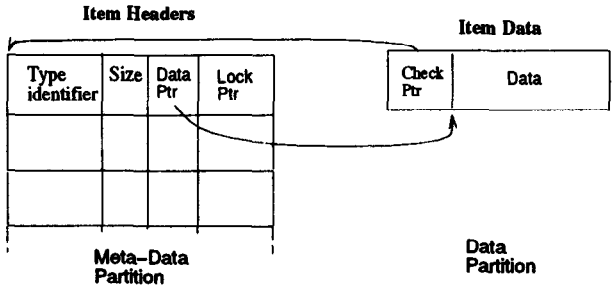

Figure 2: Item Layout

but at the cost of inter-process communication.

3.3 Database File Identifiers and Pointers

Each database file has a unique identifier. To enable identifier allocation to be performed in a distributed manner, the identifier currently consists of a 4-byte machine address and a 4-byte local-identifier.

Locations within database files are accessed in Dalí via database pointers. The Dalí architecture has two kinds of database pointers, one of which is a direct pointer to data, which we call a DBPtr, and the other is an indirect pointer, which we call an ItemID. Both types of pointers can cross database files. Since an application may access multiple different database files, it is not reasonable to assume that a database file is mapped to a fixed point in process address space. For the above reasons, DBPtrs in Dalí contain a database file identifier and an offset within the database file. The ItemID is a DBPtr to an *item header*, which in turn contains a DBPtr to the *item data*. The structure of items is described in Section 3.4.

Database pointers are untyped pointers. Dalí also provides typed database pointers to C++ applications built on Dalí. These are implemented as a template class built on top of database pointers, and use overloading on dereferencing operations to present the appearance of an in-memory typed pointer (see, e.g., [DAG93]). Thus, one can declare a variable p to be of type DBPtr_of<Person>, and use the syntax p→name if name is an attribute of the class Person.[5] A similar template class is also provided for ItemIDs.

3.4 Items

An *item* consists of an item header and item data. Figure 2 depicts the structure of an item. Database pointers to an item (i.e., item identifiers) must point to the item header, and must *not* point to the item data directly. As in a slotted page scheme (used in most storage managers), the separation between fixed

[5] This technique, by itself, does not support virtual function invocation. The virtual function table pointers embedded in the object must also be "fixed" [BDG93].

size slots and variable sized data provided by our item structuring scheme helps with compaction and relocation.

However, our scheme differs from a slotted page scheme. First, the item header is stored in a separate partition from the data for the item, and can be provided a higher degree of protection. Second, as in the main memory storage component of Starburst [LSC92], the item header can be used to store additional information such as lock information; for example Dalí stores type identifiers for items in the item header, and can store a pointer to lock information for the item. The lock pointer helps avoid the hashing costs normally associated with lock acquisition in a disk-based database. Other data that can be stored in the item header includes access control information, compression/decompression information, and extra data pointers to implement versioning.

We shall use the term *item* in the rest of this paper to refer to a data item that is the fundamental logical unit of storage. In the case of a relational system an item typically refers to a tuple. In the case of an object-oriented system an item typically refers to an object. It is sometimes useful to have an item correspond to a portion of an object/tuple or a collection of them. Hence, we use the neutral term "item" rather than "object" or "tuple". An item identifier can be used to implement tuple identifiers in a relational system, or object identifiers in an object-oriented system. Item identifier do not guarantee referential integrity; however, referential integrity checking as done in Exodus [CDRS89] and EOS [BP93] can be built into item identifier, if desired.

3.5 Pointer Dereferencing and Swizzling

It is crucial for performance that mapping from database pointers to virtual memory addresses be done efficiently. In Dalí, each process maintains a database start-address table, which specifies where in memory each database is mapped. Given a database pointer, to get a virtual memory pointer the offset within the database that the database pointer specifies is added to the starting virtual address (obtained from the above table) of the database it specifies. The table itself is implemented as a fixed size array indexed by the (integer) database file local-identifier, along with a tree structure for identifiers larger than the size of the array. As long as database local-identifiers are small integers (which we expect will be the case) the table implementation provides very fast translation from database pointers to virtual memory addresses.

The overhead of mapping database pointers to virtual memory addresses can be reduced by swizzling the pointers when the database is mapped into virtual memory. This approach is adopted in Texas [SSP92], ObjectStore [LLOW91] and QuickStore [WD94]. Swizzling, being specific to where a database is mapped, can be problematic if different concurrently running processes map a shared database to different address locations; coordination to avoid such an occurrence becomes hard when there are multiple databases. Also, pointer swizzling requires knowledge about the data types, which Dalí does not have (although the storage design does provide space for storing type tags). Therefore, Dalí does not itself provide pointer swizzling, but it is possible to build a layer on top of Dalí to do so.

Databases can be mapped into memory "on demand", that is, the first time a pointer to the database is accessed, by modifying Dalí's database pointer translation scheme. Such a facility can help transparently access a database that is larger than virtual memory, provided it is split across multiple database files. Such a scheme is reminiscent of the technique used in ObjectStore and Texas Store to handle databases larger than virtual memory.

4 System Code Structure

The Dalí system code is modular and is organized as follows.

- **Recovery Manager.** This layer provides recoverable main-memory via logging, checkpointing and recovery services. We discuss the recovery manager in more detail in Section 4.1.
- **Memory Manager.** This layer provides memory allocation and deallocation facilities. Memory allocation is managed in several layers. The bottom layer deals with changing the size of the database, the next layer performs functions similar to malloc and above it is a layer which allocates items. Dalí supports the use of different memory allocation techniques in different partitions of a database file.
- **Concurrency Control Manager.** This layer implements concurrency control; some details of its interface and implementation are described in Section 4.2.
- **Transaction Manager.** This layer provides routines that coordinate the concurrency control and recovery managers to implement transaction begin, pre-commit, commit, and abort.
- **Item Manager.** This layer provides routines for creating, deleting and accessing items. Some details are discussed in Section 4.3.

4.1 Recovery Manager

The recovery manager is responsible for ensuring the atomicity and durability properties of transactions. It provides support for physical logging (i.e., before/after image logging), as well as logical logging (i.e., logging of operations). Recovery must be performed using logical logging for high concurrency structures such as storage allocation tables and indices.

If the database is resident in memory, database pages never need to be written to or read from secondary storage during normal processing, except for the purpose of checkpointing. Recovery related processing is then the only component in the system that deals with disk I/O, and it must be designed with care so that it does not impede the overall performance.

4.1.1 Default Algorithm

The default recovery manager of Dalí uses the main-memory recovery scheme described in [JSS93]; the key features of the scheme are as follows:

- The persistent log contains only the redo records of committed transactions; this policy minimizes I/O during logging as well as during recovery. The redo and undo records of active transactions (that is, transactions that have neither committed nor aborted) are maintained separately in main memory. Undo records of a transaction are discarded once the transaction has committed. Undo as well as redo records of a transaction are discarded once it has aborted. The undo records of a transaction are written to disk only if a checkpoint takes place (more specifically, a page updated by the transaction has to be written to disk) while the transaction is active.

- The recovery actions after a system crash make only a single pass over the log. The usual backwards pass on the log to find "winners" and "losers" and undo the actions of losers is avoided by keeping the undo log separate from the redo log.

- It is possible to repeat history if required (e.g., for functions supported by the memory manager).

- A checkpoint can take place at almost any point (namely, in any state that is logical action consistent, i.e., in the checkpointed state any action for which a logical redo or undo log is generated either must not have started or must have completed). The database is partitioned into small *chunks* (checkpointing units) that can be checkpointed separately. Interference with normal transaction processing is thereby kept small; a further optimization, applicable if hot spares are used, is described in Section 6.1.

- Our checkpointing algorithm follows a *ping-pong* scheme, that is, consecutive checkpoints are written to separate locations on disk. As a result, support from the underlying system for pages to be written atomically is not required.

4.1.2 Checkpointing Units

A database file is subdivided in two different ways. First, a database file is subdivided into units of checkpointing, as mentioned above, which we call *chunks*. Second, a database file is subdivided into units of protection, which we call *partitions*. The two ways of dividing the file are orthogonal. Partitions have to be multiples of operating system pages, since memory protection is in units of pages. Chunks should be multiples of disk pages, for efficiency reasons. The size of a chunk is a tunable parameter, and different chunks can have different sizes. In this respect, a chunk is a more flexible and general concept than a page.[6]

Different chunks in a database file can have different recovery modes. The following recovery modes are supported:

Non-recoverable. No checkpointing or logging is done for actions on such chunks. They are used to implement transient system structures such as in-memory logs, transaction tables, and lock lists. User may store other data in non-recoverable chunks; for example, indices that are recomputed on database recovery in order to reduce I/O, as suggested in [LC87].

Physical-action-fuzzy. Such chunks are checkpointed in a state where no update actions that are logged logically are in progress on the chunk, but actions that are logged physically may be in progress. This is the mode required by our default recovery algorithm.

Action-consistent. Such chunks are checkpointed in a state where no update actions are in progress on the chunk. However, uncommitted data may get checkpointed.

4.2 Concurrency Control Manager

The concurrency control manager interface supports a simple but high level interface which hides underlying implementation details. The interface functions beginAccess_CC and endAccess_CC must be used to bracket

[6] Currently databases are checkpointed as single units, but checkpointing in terms of chunks is being implemented and will be operational shortly.

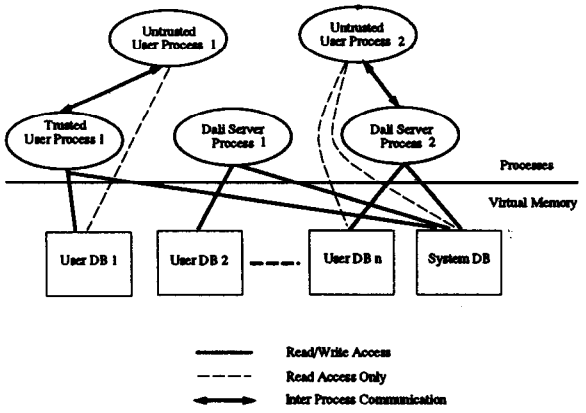

Figure 3: Example of Process Structuring

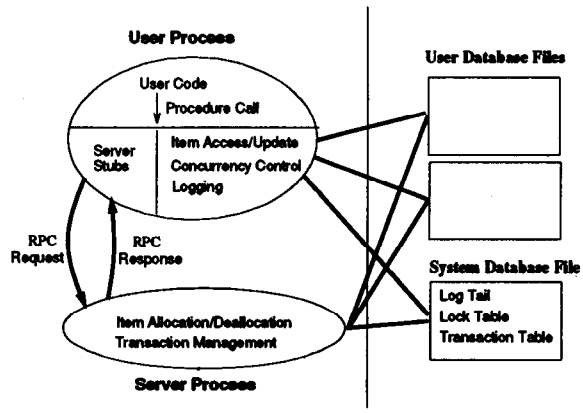

Figure 4: Direct And Indirect Access

actions. The arguments to beginAccess_CC specify the item being accessed and the mode of access. The function endAccess_CC is implicitly executed on all items at end of transaction, and need not be explicitly executed; however, if a high concurrency lock mode is used, endAccess_CC must be executed in order to release lower level latches acquired for the duration of the update action. The default implementation is based on two-phase locking, and provides a simple mechanism to introduce new lock modes with associated compatibility information for specific operations that require high concurrency (e.g., index insertion).

4.3 Item Manager

The item manager provides item-based access to the database and transparently implements concurrency control and logging. The interface may be bypassed if concurrency control and logging are to be performed explicitly. To support direct access to data, the item manager provides functions beginAccess (with ItemID and access mode as parameters) and endAccess, which can be used to bracket access to data. The interface functions get_item and put_item are built on top of the above functions, and can be used to get items from or put updated items into the database, without having direct access to the database. The item manager also implements data access paths such as hash indices.

5 The Dalí Process Interface

Access to a database file (whether a user database file or a system database file) from a user process can be done either directly to the database file mapped into virtual memory, or indirectly through a server process. The server process itself could be either a server supplied with the Dalí system, or a *trusted user process* (i.e., one trusted not to corrupt the database accidentally) that runs as a server. This is illustrated in Figure 3. The mode of access (direct or through server) is set by a flag at the time the database is opened by the user process.

Dalí offers the flexibility of choosing (at system configuration time) which Dalí functions from a user process are executed locally, and which are handled remotely by a server. Note that how the functions are executed does not affect the *interface* provided by Dalí; it only changes the *process boundaries* for the execution of the interface routines.

At one extreme, the entire functionality of the interface can be executed at a server process (via remote procedure calls). Such a process structure would be used by processes that execute on a remote machine, or that are not trusted (i.e., may have bugs). A drawback of access via a server is that interprocess communication, which is expensive in today's computer architectures, is needed for every access to the database. Further, all data has to be copied from the database to user buffers, and updated data has to be copied back into the database. Consequently, the performance of the system in this case suffers.

At the other extreme, all access to the database files (including the system database file) can be done directly from user processes. There is then no need for a server process except for recovery from crashed processes. This mode offers high performance with a higher risk of corrupting the database. Mechanisms to minimize the chances of corruption with direct access are discussed in Section 7.

We anticipate that most applications will choose a configuration in between the above extremes. Figure 4 shows a Dalí configuration where the most frequently used operations (such as reading data, obtaining locks, and writing log records) are executed in the user process (we assume the process runs on the machine where the database is resident). Less frequently invoked functions (e.g., commit/abort) are executed by a Dalí server process. Storage allocation, which involves changes to high-concurrency system structures,

is also executed at the server process.

Finally, Dalí provides to applications the ability to associate call-back functions with Dalí functions. These call-back functions, referred to as *hook functions* [HCL+90, BP93], can be used to implement triggers and maintain views.

6 Availability

Several of the target applications for Dalí require not only high performance, but also very high availability (down times of not more than a few minutes a year). Dalí features that support high-availability are described below. Mechanisms for recovery from corruption of system data-structures (Section 7) also help improve availability.

6.1 Hot Spare

Dalí supports hot spare systems (see, e.g., [GR93]). Hot spares systems receive log records from the primary system, and use the log records to keep their copy of the database synchronized with the primary; if the primary system fails, the hot spare can take over almost instantaneously. Hot spare systems are a convenient way to achieve high availability, since they can be implemented via simple changes to a storage manager without changing the internals of applications built on top. Also, by performing checkpoints at hot spares, interference of checkpoints with normal processing can be eliminated, thereby improving the overall performance of the system. Hot spares are often placed at remote locations, and communication costs may be a consideration. The optimizations carried out by the recovery algorithm used in Dalí in order to reduce disk I/O help reduce communication costs as well.

6.2 Online Schema and Software Change

Software and schema changes are inevitably required in any long-lived database system. Bringing the entire database down to incorporate such changes is entirely unacceptable in many application domains of Dalí. Dalí supports online software and schema changes in two distinct ways.

First, since Dalí supports hot spares transparently, one solution is to use a hot spare to perform the software and schema changes, while the primary continues to execute transactions. Either no updates must be allowed while the software/schema changes are being implemented, or operation logging must be used for the spare to 'catch up' – log records that refer to direct memory addresses are useless if the items referred to were modified or moved around during schema reorganization. Dalí's support for logical logging makes this feasible.

Second, some software changes can be performed while the database is up and running. Performing software changes without schema changes is facilitated by the Dalí approach of storing all system data in a system database, rather than in the private memory of a process. Dalí server processes therefore have no state stored internally, much as in the Network File System protocol [Sun88]. Thus new software can be brought up in a new server which takes over from an old server process.

7 Protection

Giving user processes direct access to the database improves efficiency by avoiding interprocess communication with a server, which may be crucial for performance in some applications of Dalí. However, direct access also increases the risk of corruption of data. We model the corruption of data by user processes which have direct access to databases and system data as occurring in one of the following two ways.

1. Software errors in user programs could result in stray writes into the database via bad pointers.

2. A user process may fail while manipulating system data structures, and leave them in an inconsistent state.

We consider below schemes to protect against such corruption. We are in the process of implementing the schemes described below.

7.1 Preventing Data Corruption

Hardware write protection on memory can be used to protect data from erroneous writes. Two variants may be used. The first is to perform updates via an unprotect-write-reprotect sequence. If an item is repeatedly updated during a transaction, changing the protection on each access would be quite inefficient, since the cost of unprotecting and reprotecting is quite significant. Hence updates are best done in a deferred fashion. The second variant is to unprotect relevant pages when obtaining a write lock and reprotect the pages at the end of the transaction.

The second variant is more efficient, but offers a lower degree of protection — the user code could erroneously write onto other data contained in a page for which protection was turned off. Further, hardware write protection does not help with corruption due to processes failing while updating shared system data structures. The next section describes schemes supported by Dalí for detecting and recovering from corruption of either form.

7.2 Detection and Recovery from Data Corruption

Corruption of items in a page can be detected by maintaining a *checksum* of an item in its header. The checksum can be verified whenever an item is read, to ensure consistency of the data, and is updated when a transaction that wrote the item commits. The checksum of an item is also verified at checkpoint time to ensure that only uncorrupted items are written to disk. If corruption is detected, transactions with write access to the page containing the item are aborted, the item is exclusive locked, and recovered by replaying the log starting from the last checkpoint of the chunk containing the item.

In order to recover system structures corrupted due to a process failure, before images of updated parts of structures are logged. The undo information is stored in *statically* allocated shared memory rather than in the process undo log. This is not only more efficient, but is also required in order to avoid a loop for updates of the undo log. If a process crashes before completing the update, the update can be undone using the before image. The before images are erased by the process after finishing the update, but before releasing semaphores.

Semaphore recovery is handled as follows. Semaphores owned by a failed process need to be released after other system structures updated by the process have been recovered [GR93]. To do this, one needs to know exactly who owned a semaphore. System semaphores provide this information, but are slow, while test-and-set semaphores, which are fast, do not provide this information. In Dalí, we implement our own fast crash-safe exclusive semaphores on top of atomic instructions such as test and set [BLS94].

8 Other Features

This section describes a few other interesting features of the Dalí main-memory storage manager.

8.1 Bulk Loading

Dalí provides special facilities for bulk loading of data, that is, the creation of a large number of items, often by loading data from another source such as a file or a database.

First, logging and checkpointing can be completely turned off for a chunk (or several chunks) of the database during a bulk load in Dalí. No other access to the chunk is allowed during bulk load, which is usually not a problem since the data is only being created. A checkpoint is taken at the end of the bulk load; the checkpoint does *not* overwrite the previous checkpoint, and hence can be aborted safely if required. The committing of the bulk load is performed by atomically noting that the new checkpoint is now the latest completed checkpoint. Performance figures presented in Section 9 demonstrate the benefits of bulk loading.

8.2 Data Compression

Compression of stored data is gaining increasing importance in today's commercial world. Compression becomes all the more important in a main-memory storage system since main memory is more expensive than disk. Due to the fixed size of a disk block, and the consequent page orientation of disk-based storage systems, there often are constraints imposed on the nature of compression that can be performed at the lower levels of the system. A main memory storage system has no such restrictions, simplifying the task of data compression.

Dalí supports data compression for the persistent log, as well as the storage of data in compressed form. Hook functions associated with the item manager functions **beginAccess** and **endAccess** can be used to compress and decompress stored data. The item manager interface automatically performs decompression on access to a compressed item,[7] and can perform compression after access to the data has ended. The separation between item headers and item data helps in this regard since the data can be moved to where space is available for decompression, without making pointers to the item invalid or requiring forwarding addresses.

8.3 Database Larger Than Memory

Dalí can be used even if the database cannot fit entirely into main memory, since the database is mapped into virtual memory. Dalí's default recovery algorithm still works correctly, but several factors need to be taken into account. Since the database pages cannot be latched into main memory, the operating system may swap dirty pages between main memory and the mapped database file at any time. If we were to map a checkpoint image directly, it would get corrupted.[8] A naive solution (currently implemented) is to copy the checkpoint image to a new file, and memory map the copy during recovery. However, with this approach, pages may be swapped to the file, and then brought back in again during checkpointing only to be written back to a checkpoint file. If we have control over what pages to flush to disk to make space for a new page (the Mach operating system provides such facilities [BGJ+92]), we can checkpoint a chunk to disk and

[7] The same mechanism is available for other operations that must modify the data before access, for instance decryption and fixing hidden pointers [BDG93].

[8] The Copy-On-Update memory mapping mode does not solve our problem since it is performed on a per-process basis.

free up the memory it occupies. We do not have such a facility in standard Unix systems, but some of this functionality can be supported through a special purpose NFS server. We are currently designing such a facility. The QuickStore [WD94] technique of using a memory mapped file as a buffer, along with memory protection to detect page faults, cannot be used when the buffer is shared by multiple processes.

In case the database size is even greater than the size of virtual memory, the database can be logically divided into multiple database files, each of which is small enough to be mapped into the virtual address space of a process. Dalí function invocations can then be directed to the appropriate process. The processes can be on multiple processors, or even in a distributed system.

8.4 Support for Large Items

Dalí does not impose a priori limits on item sizes, and items are allocated as a contiguous sequence of bytes. However, it is a bad idea to allocate large items as contiguous sequences of bytes because of the resultant storage management and fragmentation problems. As a result, large items are allocated in non-contiguous pages, and a data-structure is maintained to keep track of the pages of a large item. To support a contiguous view of a large item, Dalí provides an *item_mmap* function, which uses the mmap facility to map the pages of the item to a contiguous address range.

Recovery can be quite complicated for large items if we want to perform logical redo logging, since large items may span multiple chunks. Since chunks are written independently during a checkpoint operation, it is possible that the effects of a committed operation are reflected in the checkpoint images of only a subset of the chunks spanned by a large item. In such a situation, during restart recovery the effects of the operation on the large item (in all the chunks it spans) have to be undone, and then the operation's effects must be redone from the logical redo logs. Dalí's recovery manager maintains additional log record types for undo logging for large items.

9 Implementation and Performance

Dalí is written in C++, and uses only standard operating system functionality. Most of the features of Dalí described in this paper have been implemented.[9].

We ran a series of tests to gauge the performance of the Dalí storage manager and to understand the costs better.

[9]Universities interested in obtaining a no-fee license for the system should send mail to sudarsha@research.att.com.

Where	Lookups/trans.	Lookups/sec.
Local	1	26,110
Local	25	116,000
Local	100	130,000
Remote	100	24,100

Table 1: Lookup Performance

9.1 Performance Benchmark

We use a phone number lookup/update application, which is representative of several applications we have seen for a main-memory database system, as our benchmark application. Our benchmark application models typical environments in which main-memory databases may be used, for instance in a communications network. The TPC family of benchmarks [Gra93] does not model such an environment. The application demands very low latency, forcing data to be stored in memory, and requires durability of updates, and high availability and at least rudimentary concurrency control. The application does not require query processing and other high level database functionality, and is fairly compact, so we implemented it directly on top of the Dalí storage manager, rather than on top of a full feature DBMS.

Our test database consisted of 100,000 tuples, each comprising an 8-byte key and an 8-byte non-key field. A hash index was constructed on the key field. A transaction either looked up the value of the non-key field associated with a specified key field, or modified the information in the non-key field associated with a specified key field. We did not use a hot spare during our measurements, because our initial implementation of hot spares require performance and functionality improvements before numbers would be meaningful.

The test was run on a two-processor SPARC 10 model 51 running SunOS 4.1.3, with the database entirely in memory. All but the last test were run on a single processor. All numbers presented in this section are in terms of elapsed time on a lightly loaded system, unless otherwise noted.

The requests on the database would be generated at remote sites in a real system. Our tests do not include the cost of communication associated with remote generation of requests, and are meant to quantify the performance of the core functionality of Dalí. Instead of using full-fledged concurrency control, the implementation used shared semaphores since they suffice for the application.

9.2 Results of Performance Study

Table 1 shows the throughput on lookup queries, varying the number of lookups run as one transaction. When doing one lookup/transaction with all processing performed locally, we obtained 26,110

Updates	Ops./trans.	Ops./sec.
100 %	1	720
100 %	100	1,900
10 %	1	7,140
10 %	100	18,180

Table 2: Update Performance

lookups/second. Typically we expect several requests to be received at the same time, so we can combine multiple lookups into a single transaction. Throughput then increases significantly to 130,000 lookups per second. Using multiple lookups in a transaction amortized the cost of transaction startup between the lookups.

The results in the row labeled 'remote' are for the case where only transaction begin/commit are performed remotely. Even though reads and concurrency control are performed locally,there is a significant reduction in performance, bearing out our claim that direct access to the database is important for high performance.

We also ran an experiment using two processors simultaneously performing lookups (but not updating) a database. We were able to obtain 98,000 lookups/second each, for a total of nearly 200,000 lookups per second, as compared to 130,000 lookups/second using only one processor at a time. The loss of speed on each processor is primarily attributable to contention on the transaction table (guarded by a test-and-set based semaphore).

Table 2 shows the throughput rates when updates are present; a single process executed all transactions serially. For this test, we used group commit, with 100 update requests per log flush. For both the 10% and the 100% update case we found the system to be I/O bound, the culprit being the I/O for logging. The fact that the recovery manager in Dalí does not write undo log records to disk (except during checkpointing) is therefore important for performance. The difference between running one update per transaction (720 updates/second) and 100 updates per transaction (1,900 updates/second) is due to the transaction-start and commit log records generated by the recovery manager that must be written to disk.

We expect the performance on updates to be considerably improved once we fix the following problem. We used regular Unix files for the redo log; every log flush ended up performing (at least) two disk writes — one for the data and one to update the inode of the file. This in turn resulted in seeks occurring on every write, limiting performance to around 70 seeks per second, or at most 35 log flushes per second. We are currently implementing support for raw disk partitions to hold the redo log, which would eliminate the inode update; log flushes would then not require any seeks, improving performance.

Load type	Where	Elapsed Time (secs.)
Normal	Local	320
Bulk	Local	23
Bulk	Remote	540

Table 3: Loading times

Table 3 illustrates the benefits of bulk loading. In our test, we created the database (100,000 tuples) both in bulk load mode (no logging during loading, and a checkpoint after loading) and normal mode (100 transactions, each of which loaded 1000 tuples with logging on, and a checkpoint after loading). Bulk loading was an order of magnitude faster than normal loading since the I/O to disk had been greatly reduced. The tradeoff between bulk loading and logging is primarily determined by how much delay can be tolerated for committing updates, and the relative size of the checkpoint as against the log that is generated.

The numbers for remote bulk loading are for a configuration where a server process handled all requests except read/write of items. The time difference between local (23 seconds) and remote execution (540 seconds) during bulk loading again underlines the arguments made earlier in the paper that avoiding the use of server processes has significant performance benefits.

As argued in [JSS93], the recovery algorithm we use is quite fast. It required only 3 seconds to replay a log containing 100,000 records, each with a total of 44 bytes of data. We expect that a significant part of the log was cached in the file system buffers before recovery started, so the only costs measured are the cpu costs.

10 Conclusion

We have provided an overview of Dalí a high-performance storage manager for main memory resident data. Dalí delivers very high transaction rates with low latency, and yet provides high availability and data protection at least as good as in a disk-based system. Through modular design and multiple levels of interfaces, Dalí facilitates the construction of applications which select the functionality and performance they desire.

MM-Ode, a main memory version of the object-oriented database Ode, is already operational on top of Dalí. We are in process of implementing a relation manager on top of Dalí, and are alo considering porting an SQL query engine to run on Dalí. Also planned is a version of Dalí tailored for multiprocessors such as the NCR 3600, which have board level shared memory, and high speed interconnects between boards.

Acknowledgements

We would like to thank Alex Biliris, Narain Gehani and Thimios Panagos for useful discussions; Mark Plotnik and Jacques Gava for providing information about operating system features; and Shaul Dar for numerous suggestions that helped improve the presentation significantly.

References

[AG89] R. Agrawal and N. H. Gehani. Ode (object database and environment): the language and the data model. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, May 1989.

[BDG93] A. Biliris, S. Dar, and N. Gehani. Making C++ objects persistent: Hidden pointers. *Software - Practice and Experience*, 23(12):1285–1303, December 1993.

[BGJ+92] D. L. Black et al. Microkernel operating system architecture and Mach. In *Procs. of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, April 1992.

[BLS94] P. Bohannon, D. Lieuwen, and S. Sudarshan. Fast, crash-safe semaphores. Submitted for publication, 1994.

[BP93] A. Biliris and T. Panagos. EOS user's guide, release 2.0.0. Technical report, AT&T Bell Labs, 1993. BL011356-930505-25M.

[CD+94] M. J. Carey, D. J. DeWitt, et al. Shoring up persistent applications. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, 1994. To appear.

[CDRS89] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Storage management for objects in EXODUS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts and Databases*. Addison-Wesley, 1989.

[DAG93] S. Dar, R. Agrawal, and N. H. Gehani. The O++ database programming language: Implementation and experience. In *International Conf. on Data Engineering*, April 1993.

[GL92] V. Gottemukkala and T. Lehman. Locking and latching in a memory-resident database system. In *Proc. VLDB*, pages 533–544, August 1992.

[GLS94] N. Gehani, D. Lieuwen, and S. Sudarshan. MM-Ode: A Main-Memory OODBMS. In preparation, 1994.

[GMS92] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[Gra93] J. Gray, editor. *The Benchmark handbook for database and transaction processing systems*. Morgan Kaufmann, 2 edition, 1993.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, pages 143–160, March 1990. Also available as IBM Research Report RJ7278, San Jose, CA, Jan. 1990.

[JSS93] H.V. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *Procs. of the International Conf. on Very Large Databases*, 1993.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[LC87] T. J. Lehman and M. J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, pages 104–117, 1987.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *CACM*, 34(10):50–63, October 1991.

[LSC92] T. Lehman, E. J. Shekita, and L. Cabrera. An evaluation of Starburst's memory resident storage component. *IEEE Trans. Knowledge and Data Engineering*, 4(6):555–566, December 1992.

[PPTT91] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, a distributed system. In *Proc. of the Spring 1991 EurOpen Conf., Troms*, pages 43–50, May 1991.

[SGM90] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Trans. Knowledge and Data Engineering*, 2(1):161–172, March 1990.

[SSP92] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An efficient, portable persistent store. In *Proc. Fifth Int'l. Workshop on Persistent Object Systems*, September 1992.

[SZ90] E. Shekita and M. Zwilling. Cricket: A Mapped Persistent Object Store. In *Proc. of the Persistent Object Systems Workshop*, Martha's Vineyard, MA, September 1990.

[Sun88] Sun Microsystems. *Network File System: Version 2 Protocol Specification*, May 1988.

[WD94] S. J. White and D. J. DeWitt. Quickstore: A high performance mapped object store. In *Procs. of the ACM SIGMOD Conf. on Management of Data*, 1994.