Construction of a Main Memory Database Testbed

Chris H. Corti
Margaret H. Eich
Department of Computer Science and Engineering
Southern Methodist University
Dallas, TX 75275

Abstract
Performance of the shadow-copy update method in a memory-resident database environment has previously been studied analytically and by simulation. However not all of the several possible shadow memory architectures have been investigated thoroughly. This article describes a transaction processing testbed facility that has been constructed to examine performance of database operations using actual shadow memory implementations.

1 Introduction
In [5] it was demonstrated analytically that, in the context of a main memory database (MMDB) environment, certain deferred update approaches using a shadow-copy method [14] outperform the immediate update technique in terms of time and space utilization. Shadow copying refers to storing an after image (AFIM) in a buffer local to the transaction until the transaction commits. At that time, updates are applied to the primary database. The term "shadow memory" has been given to the shadow-copy buffer [8]. Deferred updating has the advantage that, unlike the immediate update method, before images (BFIMs) are not needed for UNDO, thus eliminating main memory accesses to read the BFIM and likewise removing the necessity of storing a BFIM in the log. Furthermore, only clean data will be written to the secondary database during the main memory checkpoint process.

A new technique for performing the insert operation was also introduced in [4]. Known as UNDO-Free Insert (UFI), this method permits storing the after image of an entire record directly into some unused slot in the database. The insert AFIM is subsequently written on the log, and, using the current database management system (DBMS) update algorithm, free space data is modified to reflect the addition of this record. In case of transaction abort, the inserted record does not have to be removed since free space data will be unchanged at the end of abort processing.

While preliminary results tend to favor deferred updating, there are several shadow memory architectures and update strategies to consider. Certain designs have already been eliminated from consideration due to their poor performance in the previous analytic experiments of [5]. Of those remaining, however, further examination is necessary to provide a broader and more realistic performance assessment under various transaction mixes and loads. This paper describes the design and construction of a transaction processing system testbed specifically for this purpose.

Section 2 describes the overall MMDB architecture as well as the basic shadow memory designs and their refinements. In Section 3 the major testbed modules and operating system support for those modules is presented. Operand relations used by the testbed and the input stream of commands are discussed in Section 4; there is also a brief indication of testbed experiments that will be performed. Section 5 concludes the paper.

2 System Architecture
This section provides a general overview of the main memory database architecture under study. Following this the specific shadow designs implemented in the testbed are discussed.
2.1 MMDB Architecture

Figure 1 depicts the model MMDB architecture assumed in this work. It is patterned after the MARS design suggested by Eich [8] and is general enough to have applicability to any data model, e.g., relational, object-oriented, etc. As the figure suggests, the MMDB is a backend database machine consisting of two processors, main memory, a stable memory region, and disks for the log and archive memory. The Database Processor, DP, receives commands from the host and performs normal transaction processing. When a transaction commits or aborts, the DP signals the Recovery Processor, RP, which executes the necessary terminating recovery activity, allowing the DP to asynchronously initiate another transaction. Commit activity involves copying AFIMs from the shadow to MM and to the log buffer, then freeing shadow areas occupied by the after images of a terminating transaction. When the RP completes this work on behalf of a transaction, it informs the DP so that any items locked by the transaction can be released. In the absence of tasks from the DP, the RP performs fuzzy checkpointing [11] of completed main memory (MM) updates. The primary database is considered to be completely memory-resident.

The stable memory, SM, region contains the shadow memory. This region is actually nonvolatile semiconductor memory which also contains the log buffer and possibly other recovery-related data, such as a checkpoint bit map. We assume that the SM is made stable by implementing the proper recovery algorithms. The log buffer is assumed to be large enough to hold the updates of all active transactions. Secondary storage is provided for the log itself. A more complete discussion of transaction processing and recovery activities in this model can be found elsewhere [8].

2.2 Shadow Memory Designs

Three candidate shadow memory designs have been chosen based upon results of previous performance model. These are (1) a set-associative shadow, (2) an SRAM shadow, and (3) a fully associative approach. Each will be briefly described in the following subsections. Throughout this discussion, it is assumed that each shadow entry has the format: <Transaction ID, MM Logical Address, AFIM>.

2.2.1 Set-Associative Shadow

The set-associative approach is realized by dividing the SM region into n separate memory banks, as shown in Figure 2. Each updated item’s main memory logical address will be mapped onto a set of n physical shadow addresses instead of just one. Whenever a set is requested, all addresses located in the set are accessed simultaneously.

Transforming a database logical address into a shadow physical address will be accomplished with bit-selection hashing [15]. Although database applications typically exhibit less locality than conventional programs [6], bit string selection should be made in such a way that the effect of attribute locality [3] is avoided. This would preclude the choice of only high order bits in case updates might be to all records of a page. Alternatively, choosing only low order bits could cause set overflow assuming the same attributes occur at the same offsets within each page. One strategy, proposed in [7], is to choose bits in the middle of the logical address.

To illustrate the shadow access strategy, we assume that the MM logical address is composed of a partition (or segment) number, page number, and offset. Each address, A, is mapped onto a set of n physical locations by a function

\[ h(A) = \{p_1, p_2, \ldots, p_n\} \]

Physical addresses correspond to the same relative ad-
address within the $n$ separate memory banks. Whenever the hash table is consulted, all addresses in the set are examined simultaneously in elapsed time of one shadow access. A successful search indicates a match of the query address with one already in the shadow; data is thereby made available for a retrieval operation or overwritten during an update. An unsuccessful search during a retrieval operation means that the data will be obtained from MM by a dual address translation process. For a modify operation, an unsuccessful search implies that the item has not been previously modified and, if an empty slot exists, it is overwritten during the write cycle.

Synonyms occur when two or more logical addresses map to the same set. Access time degradation due to synonyms, typical of normal hash schemes, is avoided with the set-associative approach as long as not more than $n$ updates per set are attempted. Beyond that, a set overflow condition results which must be handled differently. Set overflow handling and the particular bit string to be hashed are factors to be studied with the testbed experiments.

2.2.2 SRAM shadow

The SRAM shadow is implemented as a static hash table, and relies entirely on software to provide efficient key-to-address transformation and collision resolution. As illustrated in Figure 3, chained bucket hashing [12] has been implemented in the testbed. Each cell of the hash table is the head of a linked list containing all records with keys that map to that location. As chains become longer, searching linearly along a chain degrades performance. To enhance lookup speed, items in a chain are kept in sorted order by MM logical address. Furthermore, a reduction in the pointer-to-data ratio is obtained by placing more than one AFIM per node. Items within each node are, of course, kept in sorted order; if only two items are maintained per node, preserving the order entails at most a single exchange when inserting or deleting at a bucket.

In an actual implementation of the shadow, if the required memory size were somewhat underestimated, adding more memory would be relatively inexpensive while requiring only minimal software adjustment. Thus if overflow were a problem in an actual implementation, a larger hash table could be created.

2.2.3 Fully Associative Shadow

Content-addressable memory (CAM) contains internal logic capable of performing parallel search and parallel comparison of a key value with all words in the associative memory. With a CAM device, software overhead that would otherwise be needed for shadow address translation functions can be eliminated.

If the shadow space requirement is found to be limited, a single, specially designed associative unit may be economically feasible. Figure 4 illustrates a fully associative module intended specifically for the MMDB model [10]. Registers store the MM logical address, transaction identifier, and after image. A collection of these register sets make up the register file, the main
shadow storage element. Two bits per set indicate commit, abort, and vacant conditions.

To access the shadow, an equivalence test is performed in which each bit in the comparand register is compared with the address part of the register set. An unsuccessful retrieval produces a NO MATCH signal, signifying that the DP should take its data from main memory. To perform a shadow update, the DP places the address and transaction identifier on the address bus and the AFIM on the data bus, latching these into input and output ports respectively. Unless the same address has been previously updated, the NEXT register enables a vacant register set in which to place the update.

Transaction termination is based upon the transaction identifier. The value in the comparand register is compared with all such identifiers in the shadow. If there is a match, ABORT or COMMIT bits are set for those entries. The RP reads committed entries for a specific transaction, transferring AFIM values to the output port and invalidating the old shadow register sets for subsequent use by another transaction. Once in the output port, data is extracted, logged and forwarded to MM.

A major drawback of the fully associative shadow is its limited capacity. Word-level update granularity could exhaust shadow space quickly especially if most attribute fields are several words long, thereby requiring several entries per update. Overflow handling, performed by slower software routines, then determines efficiency of access. An alternative granularity will be indicated shortly.

2.2.4 Shadow Design Refinements

Granularity and overflow handling are two issues that must be confronted in the efficient handling of shadow updates. By granularity, we mean the size of an AFIM component in the shadow. Ideally, it should be fixed size and equal to the unit of memory access. There are two ways to accomplish this. One is to use a direct shadow, in which database after image attributes are broken down into word-length portions for storage directly in the hash table. Alternatively, one can employ an indirect shadow where a block of SRAM memory, the Shadow Heap, is set aside for the sole purpose of storing attribute-size AFIMs. In this case, a pointer is placed in the hash table indicating the beginning address of a variable-length area allocated dynamically from the heap space and holding the AFIM value. Figure 5 illustrates the indirect shadow technique. Note that a length field has been added to the shadow hash table record. This permits the use of a block move instruction to be used to access the data in the Shadow Heap. Such an instruction requires a single address translation for the entire attribute.

Shadow overflow occurs when data must be written to the shadow memory during an update operation but all locations in which the item might be placed are occupied. When granularity is at the word level, separate chaining and the fully associative method have the property that overflow results only when memory is completely exhausted. Set associative storage is said to overflow when an attempt is made to insert into a set but every slot is occupied in the set to which the incoming item hashed. When attribute granularity is combined with the use of a Shadow Heap, overflow can also result from expended heap space even before all available table locations are full.

To handle overflow, the following method is proposed. On an item-by-item basis, if an update causes overflow, its BFIM is stored in a separate area of stable memory until the transaction terminates, and its AFIM is written to the log buffer as well as MM. Transactions which experience overflow are placed in an ACTIVE list containing a transaction identifier and a pointer to the head of a linked list of before images. If the transaction aborts, any before images that were written will be reapplied to the database and the transaction removed from the ACTIVE list; for successful transactions, BFIM records are released and the transaction removed from the ACTIVE list prior to copying up-
dates from the shadow to MM. Recovery will require some extra processing of the log, however, the same is true if both before and after images are written there, as suggested in the previous method. By maintaining the REDO-only format of log records, the log will also be kept shorter.

3 Shadow Memory Testbed

The purpose of the testbed is to compare the performance of different shadow memory configurations and update strategies under various operating conditions and to verify previous analytic results. The testbed implements many features of an actual transaction processing system, however since low-level database operations are being studied, i.e., INSERT, DELETE, MODIFY, and READ, some higher level capabilities of an actual database management system are either omitted or present in only rudimentary form. For example, transaction preprocessing is not a part of the system; input to the testbed simulates the result of a preprocessing step in terms of the low-level commands mentioned above along with Begin Transaction (BT), and COMMIT or ABORT signals. Other researchers have concentrated on access path structures in the context of memory-resident databases [13]. Simulating ad hoc queries and access path characteristics for JOIN, SELECT, and update commands is the responsibility of the testbed input file generator. The effect of MMDB checkpointing on system performance is also omitted in this study but an evaluation of several techniques can be found in [14].

The testbed is implemented on a single-user platform consisting of an Intel 80386 processor with 8 megabytes of main memory running the IBM OS/2 Standard Edition (Version 1.2) [1] operating system. The testbed program is written in ANSI C. Code to create MM segments and to start threads requires OS/2-specific system calls, but there are few such statements in the program.

Figure 6 is a control flow diagram emphasizing key, high level modules of the testbed. After the system has been initialized, an independent thread is started to parse the input stream of database operations. This routine asynchronously supplies transactions to the system until the input file is exhausted. As a transaction is removed from the queue, an attempt is made to obtain the necessary locks and, if successful, the transaction manager begins accepting its operation requests. When a commit request is encountered, main memory and the log buffer are updated followed by housekeeping overhead to free the shadow and release locks. An abort does not result in MM or log update, but housekeeping chores are still performed.

The modules shown in this figure rely, in turn, on four basic facilities associated with the testbed which may be divided logically into services to manage transactions, main memory, shadow memory, and the log. In the sequel, we describe implementation of each service.

3.1 Transaction Management

The transaction manager monitors the transaction request queue, obtains locks for waiting transactions, and, when all locks are obtained, changes the transaction's state from waiting to ready. It then starts a thread to execute the transaction's stream of operations and resumes monitoring the request queue. Locking granularity is at the page level.

Activated transactions begin by entering a BT record in the log buffer, after which database operations are sequentially removed from the operation request stream and performed. These operations include the following.

READ This operation requires dual address translation since the item to be retrieved may be in either MM or the shadow. Two threads are started which, through calls to the main memory manager and shadow memory manager, search for a database item. If the item is discovered in the shadow, the READ routine immediately returns. Should the time to search the shadow take longer than that to obtain a value from MM, the routine waits on a response from the shadow memory manager before returning a result.
INSERT In the relation for which the insert is intended, a database page with free space must first be determined using the relation’s FirstFreePg field. After this has been accomplished, a READ operation is performed to locate that page’s most recently updated free space record which is then modified in the shadow to reflect an insertion. Next the appropriate main memory location is overwritten with the new database tuple.

DELETE A READ is performed to locate free space data, as described above, except that the intended page is known in advance. The free space record is then updated in the shadow.

MODIFY Through possibly several calls to the shadow memory manager, the entire attribute to be modified is updated in the shadow.

COMMIT Commit processing occurs in two phases: one to update main memory and the log, and the other to remove committed data from the shadow. In the MMDB model, the Recovery Processor is responsible for commit and abort processing. With the SRAM and set associative shadow, items updated in the shadow by a particular transaction have been linked on a list headed by the transaction’s identifier. As the list is scanned and AFIMs encountered, two threads are started: one to update main memory and one to write the log. When the last AFIM in a page is updated in main memory, the page lock is de-escalated so that entering transactions may obtain a share lock on it. On completing this phase, the shadow area is freed during a final traversal of the list and any locks held are released. The transaction is then removed from the ready list.

ABORT Updates in the shadow memory and all locks are released; the transaction is then removed from the ready list.

The maximum number of simultaneous transactions permitted is a system parameter, the multiprogramming level, MPL, with a default value of ten. Transactions are readied in round-robin fashion as quickly as possible. It has been suggested that in many cases serial execution would provide better throughput and response time for processing memory-resident database transactions because the overhead for concurrency control and context switching would be avoided [9]; however serial execution might not be pragmatic for real-time or long-lived transactions due to exigencies requiring preemption on the one hand, and fair scheduling on the other. Since testbed locks are obtained prior to activating a transaction, running transactions never block due to lock contention; deadlock situations never arise either. Transactions may, however, be preempted by requiring more than one time slice. Currently, all transactions run at the same priority level.

3.2 Main Memory Management

The main memory manager implements a data model consisting of relations containing fixed-size records. Records are composed of several fields each of which is either an integer, or character data of predetermined length. The testbed creates three specific relations based upon the Wisconsin Benchmarks [2] for use in the shadow update experiments. These relations and experiments will be described later in Section 4.

Retrieval, insert, delete, and modify operations are supported by the main memory manager. Individual record fields, i.e., relation attributes, may be retrieved or modified, including free space flags which are considered part of the database. The one occasion where a record-level update occurs is during the UFI (insert) operation. Until a transaction commits, the only operations against the primary database are retrievals and UFI updates.

Figure 7 illustrates the memory organization maintained by the main memory manager. OS/2 actually uses segmentation unless paging is enabled; however the paging facility is not well documented. The testbed therefore simulates paging by declaring all segments to be of a fixed size and maintains its own table of segment pointers, hereinafter referred to as the testbed “page table”. Relations are stored in a collection of these pages which are allocated from main memory during the testbed initialization phase. The simulated page
table for each relation is constructed as an array of pointers into an OS/2 descriptor table, the Local Descriptor Table (LDT), containing actual physical addresses. This indirection is necessary because pages are not pinned in memory, i.e., they may be swapped or moved during execution. A page-relative offset is added to develop the main memory physical address.

Each MM page consists of a header field plus free space for relation tuples. The header field, considered part of the database, indicates the record capacity of the page and holds the page's free-space bit map which may contain more bits than the number of records a page can hold. Together these two fields ensure correct operation of insert and delete functions. Tuple sizes are different in each of the three testbed relations but are of fixed length. Within a page, records are placed contiguously and do not overlap pages.

3.3 Shadow Memory Management

The shadow memory manager implements the SRAM and set associative models described in Section 2.2. Performance for the associative shadow is developed from the set associative model by modifying statistical monitoring techniques rather than changing basic data structures. For each model, two different testbed architectures are built so that update granularity at both the word and attribute level may be studied.

Shadow memory management includes support for search, retrieval, insertion, and modification of shadow updates. Reclamation of freed shadow areas, possibly requiring shadow reorganization, is also a part of memory management tasks. The Database Processor invokes services of the shadow memory manager during execution of regular transaction operations while the Recovery Processor utilizes its facilities only during commit processing.

The memory organization of the set associative shadow was illustrated in Figure 2. In the testbed, memory banks are constructed as separate arrays; a set is determined by the result of a hash on certain bits in the MM address. This value provides the common element of each array to access.

The SRAM shadow (Figure 3) has much more complicated data structures. Nodes must be allocated and reclaimed and, during the update process, AFIMs may have to be moved from one node to another to maintain the sorted order. The hash function is critical to performance in this architecture since every operation against the shadow requires traversing some list. A poor hash function could create long chains and would degrade this search process.

The indirect shadow has the advantage of reducing the amount of memory needed for AFIMs, i.e., for each database attribute, there is one AFIM. However, this space savings comes at the cost of additional time for allocation and deallocation from the heap resource as well as the extra level of indirection needed during shadow access. Apart from storing a pointer instead of actual data, shadow manipulations remain unchanged.

3.4 Log Management

A system log is used to record transaction updates so that the effects of a transaction may be reconstructed, if necessary, after a failure. The log itself has two parts, a buffer located in the Stable Memory region and the disk-resident log.

The log buffer consists of a number of log pages maintained as a circular queue; currently, triple buffering has been implemented. Pages normally are filled by the testbed's Recovery Processor element only during the commit phase. However, uncommitted records are also placed in the log buffer when an UNDO-Free Insert occurs. Thus, even though deferred updating is used, some log entries may represent UFI records of aborted transactions. No effort is made to detect or remove uncommitted data before writing to disk.

An attempt to write to the current log page when insufficient space remains, causes the log manager to append that page to the disk file. This I/O is done asynchronously, i.e., the log manager does not wait for its completion before continuing the logging process on the next free buffer page. The only time logging would be held up is if all buffer pages were full and I/O had not been completed on the first log page.

3.5 Operating System Support

The OS/2 environment supports multitasking with both threads and processes, threads being the basic unit of dispatch. In OS/2 protected mode, linear addressing up to 4 gigabytes is possible using virtual memory. Although OS/2 facilities for light-weight processes was appealing, the selection of this operating system was made primarily because its support for memory management is visible to an application and easily manipulated, as described in Section 3.2. If desired, extra address translation overhead can be simulated in the testbed simply by building more translation tables on top of this framework.
Multitasking and virtual memory however present a problem for obtaining accurate testbed performance statistics. We would like to have correct timing results for certain database operations; when transactions are run serially, this is no problem. However, any time transactions are interleaved, the possibility exists that after the timer has started, its subsuming thread is pre-empted. When the system scheduler reactivates the thread and the timer is stopped, elapsed time will include the time slice of the other process. To counteract this, whenever a portion of code must be timed accurately, its thread priority is temporarily raised higher than any other thread in the process.

The problem of using virtual memory is that a portion of the memory-resident database may actually be swapped to disk if memory capacity is exceeded. Swapping can be disabled globally by setting environmental variables, but this causes OS/2 to load itself into memory first leaving only about 2 megabytes for the databases, shadow memory, and testbed code. There is also an upper limit of just over 8000 Local Descriptor Table entries available to all applications; after these are expended, swapping is initiated by the system to supply more LDT slots even if memory is not exhausted. When OS/2 is memory-resident, most of these slots will be already occupied. One counter-measure to these problems is simply to enable swapping but reduce the database size until one can detect that swapping does not occur as transactions run. (Swapping is detected by querying the swapfile for time of last creation before and after the testbed runs). When swapping is enabled, OS/2 requires only about 1.5 megabytes of memory. The default database size, 1.5 megabytes, was chosen to provide the largest database that would still permit swapping to be disabled.

3.6 Timing Issues

It was pointed out in [5] that MM and shadow address translation time has an important impact on update and retrieval efficiency in the MMDB model. Main memory address translation time includes the time to consult the testbed page table plus the LDT indirection. Total access time is the summation of MM address translation and the read or write operation itself. Currently there are no plans to add auxiliary translation tables; doing so, of course, would have the effect of slowing total MM address translation.

Shadow address translation and access time differs among the three designs. We begin by describing the timing problem for the SRAM shadow which, although the most complicated, assumes no special hardware. A hash on the MM logical address locates the head of a possibly empty chain of AFIM records, or, in the case of attribute granularity, AFIM pointers. To search for one of these, a variable number of pointers has to be dereferenced, the best case being that the desired record sits at the beginning and only one indirection is required. After a record is located, if word granularity is used, the portion of the AFIM which it contains can be read immediately; otherwise, i.e., if attribute granularity is in effect, another indirection is needed to obtain the AFIM, but it represents the entire attribute.

Updating in the SRAM shadow requires the same search described above but may also entail allocating a new hash table node and linking it to the list. Ideally, when the list is searched, an empty slot can be found in an already existing node. Currently node capacity is set at two AFIM records. Thus one comparison is necessary to determine into which slot the new record must first be copied into the other slot. When attribute granularity is being used, a portion of heap space also must be allocated before the update can be made.

Set associative shadow address translation is also by hashing. In the testbed, the individual memory banks at that hash value are inspected for a match on the MM address. To simulate an actual set associative memory, only the access time, read or write, to a single bank is recorded. If attribute granularity is employed, overhead for pointer dereferencing and possibly allocation from heap space is included.

4 Testbed Experiments

A few fundamental issues concerning the shadow-copy method have yet to be addressed in the literature. Among the most important are the following.

1. The shadow area size needed to contain updates of all running transactions.

2. The shadow architecture and update algorithm that results in the best overall performance.

3. The degradation to normal performance incurred by shadow overflow handling.

This section describes the primary experiments that will be conducted to pursue these and other questions. We begin by discussing the relations against which operations are performed, and how the operation input stream is developed.
4.1 Testbed Relations

A customized database, modeled on the specification rationale of the Wisconsin Benchmarks [2], was generated to facilitate the testbed experiments. The testbed database consists of three relations; they will be referred to as ONEKTUP, TWOK2TUP, and TWOK5TUP. The first contains 1000 tuples of 100 bytes each, the second has 2000 tuples of 200 bytes each, and the last has 2000 tuples of 500 bytes each for a total size of 1.5 megabytes. These are default relation sizes, but the database can be enlarged, if necessary, up to a maximum of about 6 megabytes in the current system to satisfy experimental demands.

Each relation has two integer fields and four string attributes. The two integer fields, int1 and int2, have identical values which correspond to the record number of that tuple. The purpose of providing two integer fields is that one of them can be used as a sort key and, if desired, the other may be given a different distribution. Since access path structures are not being studied in the current experiments, these integer fields simply represent the smallest field that will be updated.

In the Wisconsin Benchmarks, each relation contains three strings of identical length. A dummy value, 'x', is given to each byte except for three significant letters in the range A,B,...,V which allow up to 10,648 unique strings. For the present study, uniqueness of the strings is immaterial but string size is not. In the testbed relations, the length of each of the first three string attributes in a relation is determined by a specific ratio of attribute size to tuple size. This fraction was modeled in the analysis discussed in [5] and its impact on the shadow space requirements and access time was measured. As in those experiments, ratios were chosen such that attribute lengths represent 10, 20, and 50 percent of the entire tuple size in the given relation; these strings are designated r10, r20, and r50 respectively.

For instance, the size of a tuple in the ONEKTUP relation is 100 bytes; therefore the r10 attribute is 10 bytes in length. In the TWOK2TUP relation, an r10 string contains 20 bytes. A fourth string, designated r111, is used to bring the relation to its specified capacity. All four strings consist of dummy values, respectively 'x', 'y', 'z', and 'f'.

4.2 Experimental Procedure

One group of experiments is intended to verify the performance model described in [5]. These tests fall into two groups: a study of shadow space requirements during update-only situations and a study of the time needed to perform each of the individual low-level operations.

To examine space requirements, a fixed distribution is used in which the probability of a modify, delete, and insert operation are given initial values of 0.05, and 0.05 respectively. As the probability of MODIFY increases to unity, the others recede equally. The effect of varying the ratio of attribute size to tuple size is examined for each relation using ratios of 0.1, 0.2, and 0.5. This test is repeated with the ratio fixed at 0.2 and a variable distribution used to investigate the effect of interaction between pairs of operations on space utilization.

The second group of experiments in the performance model were designed to study several key parameters which influence the time required in performing each low-level operation. Included in these experiments are studies of the probability of abort on the MODIFY operation, the probability of the AFIM being already in the shadow when a READ operation commences, and the effect of both shadow address translation time and shadow memory speed on MODIFY processing. All of these studies concentrate on a single operation.

Other experiments will involve running transactions that combine operations in a common mix, e.g., 70% READ, 10% each of the remaining operations. The mix can be adjusted to reflect characteristics of less-well-studied systems such as telephone switching databases in which update fields are generally small and more read operations are encountered. Transaction and update field lengths will also be varied to stress-test the shadow memory capacity and study the capability of testbed algorithms for overflow handling.

5 Conclusion

A transaction processing testbed has been constructed to investigate the shadow-copy update technique in a main memory database system. This article describes the major testbed program modules, multiprogramming system support, operand database relations, and command input stream. The testbed implements update algorithms in three different shadow memory architectures using both word and attribute level granularity; database operations are low-level and intended to be independent of a data model. A variety of experiments will be conducted to determine the architecture which provides the best overall performance, including shadow overflow handling, and the shadow capacity needed to minimize the occurrence of overflow. Results of this investigation will be forthcoming.
References


