MULTIPROCESSOR MAIN MEMORY TRANSACTION PROCESSING

Kai Li
Jeffrey F. Naughton

CS-TR-159-88

June 1988

Multiprocessor Main Memory Transaction Processing

Kai Li and Jeffrey F. Naughton
Department of Computer Science
Princeton University

Abstract

We describe an experiment designed to evaluate the potential transaction processing system performance achievable through the combination of multiple processors and massive memories. The experiment consisted of the design and implementation of a transaction processing kernel on stock multiprocessors. We found that with sufficient memory, multiple processors can greatly improve performance. A prototype implementation of the kernel on a pair of Firefly multiprocessors (each with five 1-MIPS processors) runs the standard debit-credit benchmark at over 1000 transactions per second.

1 Introduction

Recently the database community has been examining both large main memory machines and multiprocessors as a means to improve system performance in transaction processing applications. In this paper we describe an experiment designed to evaluate the potential improvement in system performance achievable through the combination of multiple processors and massive memories. The experiment consisted of the design and implementation of a transaction processing kernel on stock multiprocessors. We found that with sufficient memory, multiple processors can be used to greatly improve system performance.

Moving from disk resident databases to memory resident databases clearly facilitates the accessing and modification of the data. The problem, of course, is making this in-memory database persistent and stable. To achieve persistence and stability without sacrificing performance, the transaction processing kernel incorporates three main features:

1. The use of simple data structures for the database;
2. The parallel preparation, processing, logging, checkpointing, and acknowledgement of transactions;
3. The grouping of transactions to reduce the cost per transaction of such operations as disk writes and inter-process communication.

These features allow the kernel to achieve high performance, even when implemented on relatively slow processors.

When attempting to use parallel hardware to speed up an application, it is critical to focus on the time consuming portion of the application. Using parallelism to speed up a small fraction of the application will have a only a small effect on system performance.

A transaction that merely reads and modifies a few in-memory records from some well-chosen data structures spends very little time actually executing. The bulk of the time is spent by the system on behalf of the transaction, doing such things as disk writes (for logging and checkpointing) and system calls (to receive the transaction, send an acknowledgement, synchronize with other processes, perhaps get the current time for a timestamp, and so forth.) Hence it is precisely these time consuming activities, rather than the execution of the transactions themselves, that our implementation parallelizes and amortizes.

We implemented a prototype version of TPK on DEC Firefly multiprocessors [TS87] to see to what extent this approach to parallelism would speed up the application. We found that in multiprocessor configurations, TPK achieves "superlinear" speedup, that is, the performance increase over a uniprocessor configuration exceeds the number of processors in the system. Superlinear speedup of a given algorithm is of course theoretically impossible; however, superlinear speedup of a system, through more efficient use of system resources, is not.

The experiments presented here were performed at the DEC Systems Research Center, and tested the following three configurations:

A. a uniprocessor with two disks,

B. a shared-memory multiprocessor with a two disks,

C. two shared-memory multiprocessors (each with a disk) connected by a loosely-coupled link.

In configuration C, TPK produced a maximum throughput of over 1000 debit-credit transactions per second, indicating that on shared-memory multiprocessors with large enough main memories, a simple transaction processing system can produce very high performance.

In related work, there are a number of papers presenting multiprocessor transaction processing and recovery architectures for traditional, disk-based database systems. (For a survey, see [HR83]). However, for memory resident databases the set of performance tradeoffs is different. Perhaps the most salient difference is that a transaction that reads and writes several disk records will last much longer than one that reads and writes only in-memory records. For such disk-based transactions, the overhead of system calls and log writes is a much smaller factor in system performance than it is in a system such as ours.

Surveys of main-memory transaction processing schemes include [Eic87, SG86a, SG87]. The schemes surveyed in [Eic87] and [SG86a] for the most part assume special purpose or dedicated hardware. (Similarly, the individual schemes proposed in [Eic86] and [LC87] both assume some amount of special purpose hardware.) The schemes surveyed in [SG87] do not assume special hardware, but also do not consider multiprocessors. While our scheme makes use of a lot of

memory and multiple processors, we do not assume special purpose hardware — in fact, our prototype implementation is on an unmodified general purpose multiprocessor.

Our design draws upon some ideas from several earlier proposals. Like the system proposed in [DKO*84], our system uses group commits. Our system uses asynchronous fuzzy checkpoints, similar to those proposed in [Hag86]. However, that system cannot support pre-committed transactions without special purpose hardware. By combining the use of extra memory and the fact that only one transaction is active at a time, our system can use both pre-committed transactions and asynchronous fuzzy checkpointing, without the use of special purpose hardware such as a stable segment of memory.

2 Overview of TPK

TPK is a kernel that supports general transaction processing with coarse-grained parallelism. It does not include any detailed implementation of the communication protocols between transaction processing and client's terminals. The generic design of the transaction processing kernel (TPK) consists of five[1] kinds of threads (a thread is a lightweight process):

- *input.*

  The input thread receives transactions from clients, prepares them, and submits them to an input queue.

- *execution.*

  The execution thread takes transactions off the input queue, processes a group of transactions, and inserts log records in a log queue.

- *logging.*

  The logging thread deletes records from the log queue, writes these records into the disk-based log, and inserts processed log records into the output queue.

- *output.*

  The output thread deletes records from the output queue, sends the results to clients, and sends log records to the checkpoint thread.

- *checkpointing.*

  The checkpoint thread deletes records from the checkpoint queue and performs fuzzy checkpointing.

A diagram of TPK appears in Figure 1.

Given enough processors, all five kinds of threads in TPK run concurrently. On the other hand, as the parallelism in TPK is coarse-grained, TPK can be run efficiently on a uniprocessor, time-sharing between the threads.

[1]The version of TPK described in [LN88] consisted of four kinds of threads, including a combined execution-logging thread. The suggestion to split this thread was made by Garret Swart.

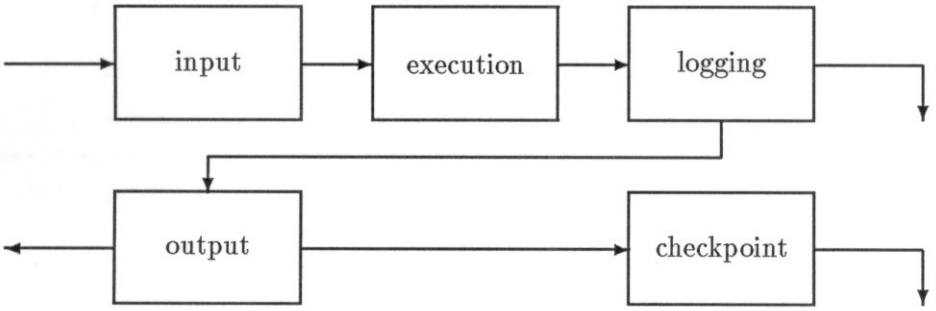

Figure 1: System diagram for TPK.

Transaction Input and Output

The input thread accepts transactions and inserts them in an input queue. The goal of the input thread is to free the execution thread from much of the overhead involved with the preparation of transactions.

Once a transaction has been prepared (that is, once a record with the relevant information corresponding to the transaction has been built), the record for the transaction is inserted in the input queue.

The input queue is shared between the input thread and the execution thread. Because of this, insertion into and deletion from the transaction queue are both critical regions, and are protected by a semaphore. When the execution thread wishes to receive new transactions, it must first aquire the lock on the input queue.

In order to minimize the contention between the execution thread and the input thread, after acquiring the lock on the input queue, the execution thread deletes all transactions currently waiting in the queue, not just the first. Hence if there are on average k transactions in the input queue each time the execution thread accesses the queue, the overhead per transaction as seen by the execution thread is reduced by a factor of k.

The output thread is responsible for deleting finished transactions from the execution thread's output queue, for delivering these transactions to the checkpoint process, and for sending responses to clients. The goal of this thread is the same as that of the input thread — to free the execution thread from overhead.

If either transaction input or output becomes the bottleneck in the system, both the input thread and the output thread can be replicated to take advantage of multiple processors to increase performance.

Transaction Execution

The execution thread takes transactions from the input queue, executes them, prepares log records, and inserts them into the log queue.

After retrieving a group of transactions from the input queue, the execution thread inserts them into a private queue. Both the deletion and insertion of transactions are done as a group, that is, they are accomplished by a single delete and a single insert operation. The execution thread then processes the transactions in order.

```
BEGIN
    copy all records to be modified, for use in case of abort;
    execute transaction;
    IF execution was successful
         insert changes in log buffer;
         mark success;
    ELSE
         restore modified records to original values;
         mark failure;
    END;
END.
```

Figure 2: Schema for processing a generic transaction.

The schema for a generic transaction is given in Figure 2. First, the transaction makes copies of all records to be modified. Next, the transaction performs the database reads and writes necessary to execute the transaction. If for any reason the transaction execution fails (for example, there is an attempted division by zero or a reference to a record for which it does not have the proper permission), the original copies of any modified records are restored, and the transaction is marked as failed.

If all changes are completed without the transaction raising an exception, then copies of the changed records are inserted into the current log buffer. After the changes have been recorded in the log buffer, the transaction is marked as successful.

Note that because only one transaction is in the read-modify-write phase at any given time, there is no need for record locking. This is not to say that only one client can access the database at once; rather, all clients must access the database through TPK, which is responsible for serializing transactions. As documented in Section 4, TPK achieves very high transaction rates without concurrent transactions. This is possible only because the data upon which the transactions execute is memory resident.

Logging

The logging thread removes records from the log queue and writes them to stable storage on disk. To avoid a log disk write per transaction, we use the well-known technique of *group commits* [DKO*84]. In group commit, multiple log records are buffered and then written to the disk as a group, thus amortizing the cost of the disk write over all the transactions in the group.

In our implementation of group commit, the time at which to write the log buffer to the disk is controlled by two parameters, `TransactionsPerGroup` and `TimePerGroup`. A group record is written to disk either when it contains `TransactionsPerGroup` transactions or when more than `TimePerGroup` milliseconds have passed since the first transaction log was written into the record. The use of a group timer to improve response time (and even system throughput) has been discussed in [GK85, HSL*87].

After a group of transactions has committed, the corresponding records are added to the output queue (for use by the output thread in responding to clients) and to the checkpoint queue (for use by the checkpoint thread to produce a checkpoint.)

Correctness and Efficiency

At this point the reader may have two concerns. The first is correctness, and the second is efficiency. As we do no locking, correctness is an issue. However, since in our system only one transaction is actively modifying data at any given time, it is not necessary to obtain locks before reading and writing records. This is consistent with the observation in [GLV84] that in a main memory database, it may be possible to eliminate concurrency control by serializing transactions.

Note that we are not eliminating the possibility of multiple clients concurrently accessing the database. Instead, we require that any client wishing to access the data does so through TPK, which is responsible for serializing these accesses.

Another correctness concern arises because, with group commit, transactions may be run before preceding transactions have committed. This raises the possibility of another transaction reading "dirty data," that is, data written by some transaction that has not yet committed. Because we have only one transaction execution thread, this is not a problem, which can be seen as follows.

Suppose that a transaction t_1 writes a record, and that another transaction, say t_2, later reads that record. Because transactions execute serially in TPK, t_2 cannot be a member of an earlier commit group than t_1. This leaves two cases. If t_2 is a member of a later commit group than t_1, then t_2 will commit only if t_1 commits. If t_2 is a member of the same commit group as t_1, then t_2 will commit if and only if t_1 commits. In either case, the outcome is correct.

The efficiency issue arises because in our system, only one transaction can be modifying data at a time, that is, the actual execution of the transactions is serialized. The parallelization of transaction executions is a critical issue if transactions are long-lived; however, transactions that simply read and modify a few in-memory records (those our system is designed for) are extremely short-lived. The bulk of the time required to process this sort of transaction is consumed by overhead such as writing the log record and doing the interprocess communication necessary as the transaction progresses through the system. It is this overhead that is parallelized and amortized in TPK.

Checkpointing

For simplicity, in this paper we assume that if one processor in the system crashes, the entire system crashes.

The goal of the checkpointing thread is to periodically produce a checkpoint of the database on disk. After the snapshot has been recorded, all log entries corresponding to transactions whose effect is present in the checkpoint can be ignored.

We maintain two copies of the database, DB, and DB'. DB is the in-memory copy of the database upon which the transaction execution thread operates. DB' is an in-memory copy used by the checkpoint thread. To incorporate checkpointing, the transaction execution thread, upon committing a transaction, inserts the transaction's log record into the checkpoint queue (in addition to writing the record to disk).

Periodically the checkpointing thread empties the checkpoint queue and processes the records so obtained. By having the checkpoint thread delete transactions from the checkpoint queue in groups, the cost of this interaction between the checkpoint thread and the transaction execution thread can be reduced to low levels.

The checkpointing thread has two distinct phases. During the first phase, the *update* phase, the checkpointing thread repeatedly deletes a batch of records from the checkpoint queue, and installs the changes specified by the log records into DB'. By *installing changes* or *applying a log* we simply mean copying the value of a record in the log record to the corresponding record in DB'. After processing some number of updates, the checkpoint thread stops processing updates and switches to the second phase.

During the second phase, the *writeout* phase, the checkpointing thread writes out the entire copy of DB' to disk. After it has done so, the checkpointing thread notifies the transaction execution thread of the last log record whose changes were installed in the checkpoint of DB'. Then the checkpointing thread begins the next update phase.

While the checkpoint thread is in the writeout phase, log records will accumulate in the checkpoint queue as the transaction execution thread continues. As the checkpoint thread does no log writes or record copies during the update phase, it processes log records much faster

than they are produced by the execution thread, so the checkpoint thread will catch-up during the next update phase.

Recovery

This checkpointing thread divides time into a series of checkpoint intervals. We will use DB'_i to refer to the copy of DB' that is written to disk during interval i. Similarly, we will use L_i to refer to the log records that are installed in DB'_i during the update phase of interval i.

If the system crashes while the checkpoint thread is in the update phase of checkpoint interval i, recovery is simple. The system first reads the copy of DB_{i-1} from disk into DB', then makes DB a copy of DB'. Then the system must apply the log to DB, to obtain the state DB held immediately prior to the crash. At this point the transaction execution thread can be restarted, and the checkpoint thread can be restarted, checkpointing the log records that were just applied to DB.

If instead the system crashes during the writeout phase of checkpoint interval i, a subtlety arises. It is possible that the on-disk copy of DB'_i could be left in an incomplete and corrupted state. This problem can be avoided by allocating enough space on disk to hold the checkpoint of DB', plus one extra disk block. If the database fits into n disk blocks, then we allocate the $n+1$ disk blocks 0 through n for storing copies of DB'.

We manage these $n+1$ disk blocks as a circular buffer as follows: when the first checkpoint of DB' is written, block k of DB' is written to block k on the disk. On the second checkpoint of DB', block k is written to block $k-1 \bmod n+1$. In general, suppose that on checkpoint i, block k of the database is written to block j on the disk. Then checkpoint $i+1$, block k of DB' is written to block $j-1 \bmod n+1$ on disk. This technique is mentioned in [Hag86] and is called "sliding monoplexed backups" in [SG87].

Suppose we are in checkpoint interval i, and while writing block k of DB' to physical block j on disk, the system crashes. Physical block j may be corrupted. However, we will have good versions of blocks 1 through $k-1$ of DB'_i in physical blocks $j-k \bmod n+1$ through $j-1 \bmod n+1$, and good versions of blocks k through n of DB'_{i-1} in blocks $j+1 \bmod n+1$ through $j+1+(n-k) \bmod n+1$.

Hence the recovery process can read these good blocks to produce a hybrid version of DB', in which blocks 1 through k are from checkpoint i, and blocks $k+1$ through n are from checkpoint $i-1$. This is a "fuzzy" checkpoint in that it may not be transaction consistent. For example, if some transaction t wrote records in block 1 and in block n, and $1 \neq n$, then in this checkpoint the records in block 1 may show the effect of t while the records in block n do not.

However, because we use record logging (rather than transaction logging) this does not create a problem. The system, upon recovery, again reads this fuzzy checkpoint into DB' and DB. Then the log is applied to DB, and the update phase of interval i of the checkpoint thread

is restarted.

In applying the log to DB and in applying the log to DB', some records from L_i will be applied to some of the first j records of the database. These records may already have had these log records installed before the crash. However, this does no harm, as applying a log (installing an update) is an *idempotent* operation. In other words, applying a log entry more than once leaves the database record in exactly the same state as applying the log record once.[2]

This checkpointing scheme is designed to separate the transaction execution thread and the checkpoint thread as much as possible. The two are independent except for a shared checkpoint queue and a message at the end of each checkpoint interval to tell the transaction execution thread which log records have been processed. In our implementation, the transaction execution thread and the checkpoint thread run on separate Fireflies, hence they have separate processors, memory, and I/O systems. Because of this, checkpointing is essentially free with respect to system throughput.

Performance

Even with this checkpointing scheme, recovery after a crash in this system is relatively slow: the system must first read an entire checkpoint of DB into memory from disk, then apply all the log records that accumulated since the checkpoint was written. This long recovery time is a problem in any volatile main-memory system, as there is no way to avoid reading the copy from disk.

However, the checkpointing and recovery portions of the system are ideal applications for the use of multiple disks, e.g. using some system such as *disk striping* [SG86b]. If d disks can be allocated to checkpointing and recovery, both reading and writing the checkpoint copy of DB can proceed at an effective rate of Td, where T is the maximum sustainable transfer rate of a single disk.

In addition to using techniques such as striping to increase the effective disk I/O bandwidth, our scheme can easily be extended to support multiple checkpointing threads, each one responsible for checkpointing a portion of the database. This will reduce the amount of time necessary to apply a batch of log records to DB'. These two techniques combine to improve checkpointing performance.

3 Prototype Implementation

The TPK design is a simple main-memory based transaction processing kernel with concurrent threads. Such a simple design can be implemented on a variety of architectural configurations.

[2] It is, of course, essential that the logged transactions be applied in the order in which they were originally run. Again, this is simple to achieve because we have a single transaction execution thread.

Our experimental results on a number of configurations indicate that on a stock shared-memory multiprocessor with enough memory, a simple transaction processing system can yield a super-linear speedup over a single processor configuration. Even on multiprocessors with 1-MIPS processors, this can generate a throughput of over a thousand transactions per second.

We implemented our prototype on DEC Firefly multiprocessors. The Firefly is an experimental shared-memory processor developed at the DEC System Research Center [TS87]. Each Firefly consists of five MicroVAX 78032 processors, each with a floating point unit and a 16 KByte cache. The caches are coherent, so that all processors within a single Firefly see a consistent view of shared memory. One of the processor is dedicated to take care of all I/O operations. When there is no I/O operation to be done, the I/O processor is treated the same as other processors. The Fireflies were connected by an Ethernet.

The prototype is written in Modula-2+, a dialect of Modula-2 developed at DEC's Systems Research Center [RLW85]. As Modula-2+ is strongly typed, it provides much better protection against database corruption due to programming errors than would a language such as C.

Since we do not have transaction terminals, the transaction input thread simulates the overhead of interacting with I/O devices by busywaiting a few hundreds of instructions before generating a random transaction. The exact amount of time to busywait is a parameter of the benchmarks, which we varied between 500 to 2000 μseconds. Our design nowhere assumes that there is only one input thread, so if the speed with which transactions can be read from external devices becomes the limiting factor in the system, the input thread can be replicated to remove the bottleneck. Similarly, the output thread busywaits to simulate the overhead due to I/O upon output.

In the prototype implementation, the execution, logging, and checkpointing threads are essentially as described in the previous sections. One implementation detail that may be of interest is our implementation of a group timer.

Checking the time since the first log record was written in the current log buffer makes use of an important feature of the Topaz operation system on the Firefly: it is possible to get the current time in a single instruction by accessing a public read-only memory location. On many other systems, this would have to be done through a system call, which would in turn require a system call for every transaction. This system call would significantly increase the time required to process a transaction. For example, in systems such as UNIX or Mach running on a 1-MIPS processor, the `gettimeofday()` system call takes about 400 μseconds.

The goal of our experiments was to find out how fast the TPK run on different architectures and to determine the degree to which multiple processors would speed up the application. With the help of Garret Swart at DEC Systems Research Center, we built a special operating system that isolated our experiments from the effects of the virtual memory system and other subsystems. The implementation of TPK supports (through run-time switches) many different configurations of processors, disks, and communication links.

We performed our experiments with the following three configurations:

A. *A uniprocessor with two disks.*

   All threads run on one processor. Logging and checkpointing are performed on separate disks.

B. *A shared-memory multiprocessor with two disks.*

   All threads share the same address space. Each thread has its own processor. Logging and checkpointing are performed on separate disks.

C. *Two shared-memory multiprocessors (each with a disk) connected by a loosely-coupled link.*

   The input thread, execution thread, logging thead, and the output thread run on the first multiprocessor. The checkpoint thread runs on the second multiprocessor. The communication between the checkpoint thread and other threads is via remote procedure call (RPC).

It turns out that in Configuration C, whether the second machine is a multiprocessor or single processor does not make much difference.

To effectively use shared-memory multiprocessors, all the threads in the TPK implementations busywait until their queues are non-empty. For the uniprocessor case, busywaiting is not reasonable because it consumes CPU time without doing any useful work, reducing throughput. Thus, each thread in the uniprocessor version blocks on a semaphore when its queue is empty, and is resumed by insertion of input to the queue.

Furthermore, structuring a uniprocessor system as a collection of communicating threads will produce overhead due to context switches between them. To remove this overhead (which is an artifact of the implementation and not of the transaction processing problem itself), in the uniprocessor version the functions of the execution, logging, and output threads were coalesced into a single thread. Hence the uniprocessor version ran with only three threads: input, execution-logging-output, and checkpoint.

The goal of removing busy-waits and extraneous context switches from the uniprocessor version was to ensure that we were comparing our multiprocessor versions with the best performance we could produce on a single processor. This is a more valid measure of the benefit of multiprocessors than would be produced by comparing a poor uniprocessor version against a multiprocessor version.

4 Experiments

In order to benchmark our system, we implemented the standard debit-credit benchmark proposed in [Ano85]. That benchmark requires the system to run debit-credit transactions against

three tables: accounts, tellers, and branches. Additionally, the system must maintain a log sufficient to produce an account history for each account in the database. A high-level description of the debit-credit transaction appears in Figure 3.

```
BEGIN
    read in transaction;
    get account;
    get teller;
    get branch;
    save account, teller, and branch for potential abort;
    validate access permission;
    check balances;
    write account;
    write teller;
    write branch;
    IF exception THEN
        restore account, teller, branch;
        abort;
    ELSE
        log transaction;
    post response;
END
```

Figure 3: The debit-credit transaction.

As the database is memory resident, it can be implemented using exceedingly simple and efficient data structures. Each of the accounts, tellers, and branches tables is represented as an array. In addition to the tables, we maintain hash tables to map from account, teller, and branch ids to array indices. Figure 4 shows the record definition for the account table. The record definitions for the teller and branch tables are the same.

Each record has four fields. The balance field is used to maintain the current balance for the account, branch, or teller the record represents. The code field holds a password used to verify

```
AccountRecord = RECORD;
    code:      INTEGER;
    account:   INTEGER;
    balance:   INTEGER;
    otherInfo: ARRAY[ 0..115 ] OF CHAR;
    END;
```

Figure 4: Record definition for account table

access rights. The account, branch, and teller fields hold the id for the account, branch, or teller represented by the record. The size of a record is made 128 bytes to satisfy the requirement of the debit-credit benchmark [Ano85].

To allow insertions and deletions of records, the system maintains a list of free array locations. Upon insertion, the next free array location is allocated, the record is inserted, and the corresponding hash table is updated to reflect the insertion. Upon deletion, the location occupied by the record to be deleted is added to the free list, the account id is set to some number that is not a possible id for an active account, and the corresponding hash table entry is marked deleted. Insertions are handled in a similar fashion.

TPK keeps two copies of the database: one for the execution thread and one for the checkpoint thread. Since the current Firefly multiprocessor does not have a lot of memory, we ran the benchmark on a database consisting of 15,000 account records, 150 teller records, and 15 branch records. A 15,000 tuple database is large enough to destroy locality of data references in each cache and to show the insight into the performance our TPK design could produce if there was sufficient memory.

To run the full debit-credit benchmark as specified in [Ano85], with 10,000,000 accounts, would require the multiprocessor to have about 1.5 Gbyte of memory. The Massive Memory Machine project at Princeton [GLV84] has modified a Sun workstation to support 1 Gigabyte of memory; in the future such machines, and machines with considerably more memory, will become commonplace. Machines that can keep at least the active portion of many database applications in memory will soon be readily available.

One of the requirements in the benchmark is that more than 95% of the transactions must respond in under one second. To time the response time of a transaction, the input thread puts a time stamp on each transaction before inserting it into the input queue. When receiving a transaction, the output thread checks the time against the time stamp in the transaction, and records the statistical information. All timing was done by using elapsed time (or wall time).

Each benchmark run consists of 50,000 debit-credit transactions. The input thread generates these transactions by randomly generating the account, teller, and branch numbers, then randomly generates an amount by which to debit or credit. The checkpoint thread writes its database copy to a disk 5 times during each run. In other words, it writes to disk after processing about 10,000 transactions. For consistency, for each set of parameters the benchmark was run several times. The results are given in Table 1.

Moving from configuration A to configuration B increases the number of processors from one to five, but improves performance by a factor of 5.83. This superlinear speedup results from a number of reasons.

In configuration B, much of the execution of the input, execution, logging, output, and checkpointing threads is overlapped. Also, in configuration B, each of the threads executes largely without interruption, thus avoiding overhead due to context switching. Furthermore, in

	A	B	C
Throughput (TPS)	77	449	1282
Avg. response time (msec)	480	505	509
Max response time (msec)	3792	1622	1228
Percent response < 1 sec	96	97	96
Group Size (transactions)	50	200	400

Table 1: System performance on debit-credit benchmark.

configuration B, as the threads execute without interruption, the system can support a much larger group size while still maintaining the one second response time. This allows the per-group overhead (most significantly, the log writes to disk) to be amortized over more transactions.

Moving from configuration B to configuration C represents adding two processors (the processor on the second Firefly doing checkpointing, and the processor on the second Firefly receiving checkpointing messages) and a second memory and I/O system. This improved performance by a factor of 2.9, an increase in performance that is again superlinear in the amount of additional hardware. This can be explained as follows.

In configuration B, although the log and checkpointing disk writes are directed to different disks, they still compete for a common disk controller. In configuration C, there is a second memory and I/O system. Thus in configuration C, the checkpoint writes never compete with the log writes. This again allowed us to double the group size while still maintaining the one second response time, further improving the maximum throughput.

Moving from configuration A to configuration C increased the number of processors from one to eight, and doubled the number of I/O and memory systems. However, it increased throughput by factor of 16.6. The main reasons for such a superlinear speedup are:

- TPK threads executed in parallel.
- Shared-memory multiprocessor substantially reduced the time spent on context-switches. (On a 1-MIPS processor, a thread context switch takes a few hundreds of microseconds. When the amortized cost of processing a transaction is about one millisecond, the cost of a context switche is significant.)
- Efficient execution of transactions allows large group size so that group-commit can further reduce the amortized cost of logging.

These three points are not independent — it is the first two points that allow us to increase the group size by a factor of eight while still maintaining the one second response time.

In order to see how can the performance of TPK be further improved, we did some more measurements to see where the time is spent in configuration C. The elapsed time (in seconds) spent in various parts of the system break down as follows:

Total Time	39.0
Input Time	38.5
Execute Time	38.5
Log Time	28.7
Output Time	12.2
Checkpoint Time	20.9

The total time is the wall clock time for TPK to run 50,000 debit-credit transactions. The input time gives the time spent in the input thread, excluding the time spent busy-waiting for transactions. Similarly, the execute time, log time, and output time give the times spent in the corresponding three threads exclusive of the time spent spinning on empty queues. The checkpoint time gives the time spent sending data to the checkpoint thread for checkpointing.

Since the TPK threads are executed in a pipelined fashion, the total time must be greater than the time for any of its subparts. We can see that the input and execute threads are both within 0.5 seconds of the total elapsed time. This indicates that further improvements in performance must come from speeding up these two threads. In the conclusion we discuss some approaches to do so.

5 Conclusion

The TPK prototype demonstrates that multiple processors can be exploited to greatly enhance the performance of main memory transaction processing systems. TPK uses multiple processors to overlap transaction input, execution, logging, output, and checkpointing, and to eliminate system overhead. It is interesting to note that in TPK only one transaction is in the read-modify-write phase at a time, so the observed superlinear speedup is not even partially due to concurrent execution of transactions.

In the fastest configuration (Configuration C) the input and execution times are both within a few percent of the total elapased time. This indicates that the performance of the prototype in its present form can not be improved significantly through the addition of more processors. If the system is to be sped up through the use of still more processors, these execution and input threads must either be replicated or broken into several pipe stages to expose more parallelism.

The way to speed up the input thread through multiple processors is clear — there is nothing in the design of TPK that prevents multiple input threads operating independently in parallel, preparing transactions and inserting them into the input queue.

Dealing with the execution thread is less straightforward. For some types of transactions, it

is possible to break the execution thread into two stages. The first stage locates the records to be modified by the transaction, so that the second stage is handed record addresses and need not do any lookups or validation.[3] Another logical approach to speeding up the execution thread is to allow multiple transactions to execute concurrently, that is, to replicate the transaction execution thread. Of course, this approach will require some form of locking to guarantee serializability of transactions. We are currently investigating both approaches to increasing the performance of the execution thread.

Acknowledgement:

DEC Systems Research Center provided us with their Firefly multiprocessors for running the experiments. Hector Garcia-Molina provided useful comments on an earlier version of this paper. Garret Swart gave us the suggestion of separating logging from execution and also helped us build a special version of the Taos operating system on which to run the tests.

References

[Ano85] Anon. A measure of transaction processing power. *Datamation*, 31(7):112–118, April 1985.

[DKO*84] David J. DeWitt, Randy H. Katz, Frank Oiken, Lenard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–8, 1984.

[Eic86] Margaret H. Eich. Main memory database recovery. In *Proceedings of the ACM-IEEE Fall Joint Computer Conference*, 1986.

[Eic87] Margaret H. Eich. A classification and comparison of main memory database recovery techniques. In *Proceedings of the Third International Conference on Data Engineering*, pages 332–339, 1987.

[GK85] Dieter Gawlick and David Kinkade. Varieties of concurrency control in ims/vs fast path. *Database Engineering Bulletin*, 8(2):3–10, June 1985.

[GLV84] Hector Garcia-Molina, Richard J. Lipton, and Jacobo Valdez. A massive memory machine. *IEEE Transactions on Computing*, C-33:391–399, May 1984.

[Hag86] Robert B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Transactions on Computers*, C-35(9):839–843, September 1986.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *Computing Surveys*, 15(4):287–317, December 1983.

[HSL*87] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phill Garrett, and Andreas Reuter. Group commit timers and high volume transaction systems. In *Proceedings of the Second International Workshop on High Performance Transaction Systems*, September 1987.

[3] This technique was suggested to us by Richard Lipton.

[LC87] Tobin J. Lehman and Michael J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 104–117, 1987.

[LN88] Kai Li and Jeffrey F. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, December 1988.

[RLW85] Paul Rovner, Roy Levin, and John Wick. *On Extending Modula-2 For Building Large, Integrated Systems.* Research Report 3, DEC Systems Research Center, 1985.

[SG86a] Kenneth Salem and Hector Garcia-Molina. *Crash Recovery Mechanisms for Main Storage Database Systems.* Technical Report CS-TR-034-86, Department of Computer Science, Princeton University, 1986.

[SG86b] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *Proceedings of the International Conference on Data Engineering*, pages 336–342, February 1986.

[SG87] Kenneth Salem and Hector Garcia-Molina. *Crash Recovery for Memory-Resident Databases.* Technical Report CS-TR-119-87, Department of Computer Science, Princeton University, 1987.

[TS87] C.P. Thacker and L.C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, October 1987.