Performance of an OLTP Application on Symmetry Multiprocessor System

Shreekant S. Thakkar
Mark Sweiger
Sequent Computer Systems
Beaverton, Oregon 97006

ABSTRACT

Sequent's Symmetry Series is a bus-based shared-memory multiprocessor. System performance in an OLTP relational database application was investigated using the TP1 benchmark. System performance was tested with fully-cached benchmarks and with scaled benchmarks. In fully-cached tests, the entire database fits inside main memory. In scaled tests, the database is larger than available memory. In the fully-cached benchmark, performance was initially limited by bus saturation. The cause was the transfer of process context from processor to processor. This was eliminated by assigning each process to a processor. Processor affinity was combined with reductions in message-passing within the database. Throughput was dramatically improved. The scaled tests were I/O bound. This bottleneck can be eliminated by connecting more disk drives, or by increasing the main memory size.

1. INTRODUCTION

Symmetry systems are bus-based cache-coherent shared memory multiprocessors running a UNIX-based symmetric operating system. Such systems allow multi-user/multi-stream applications to be ported almost transparently from a similar uniprocessor environment.

However, porting does not guarantee that applications will achieve the optimal throughput that the architecture offers. Both system and applications may have to be tuned with appropriate parameters to achieve optimal performance. This paper describes the effect of some of these parameters on system performance when running the TP1 benchmark. We do not know of any other work which describes an evaluation of internal system behavior with this application.

System bus and processor cache mechanisms were major targets of this investigation. The goal was to discover the cause of an unexpected roll-off in database performance as more CPUs were added to the system.

The background and results of the investigation are presented in the following sections:

2. ON-LINE TRANSACTION PROCESSING

Characteristics Of OLTP Systems

On-Line Transaction Processing (OLTP) applications demand rapid, interactive processing for large numbers of relatively simple transactions. They are typically supported by very large databases. Automated teller machines and airline reservations systems are familiar examples of OLTP.

OLTP is among the fastest growing segments of database applications. OLTP systems are becoming very important strategically to many businesses because they can give a competitive edge in providing service. Prominent vendors besides Sequent in the OLTP market include Digital Equipment Corporation, IBM, Stratus and Tandem Corporations.
OLTP Performance characterization

The TP1 Benchmark

The test used to benchmark OLTP database performance on Symmetry systems was the TP1 benchmark. Originally proposed in 1985 (Anon, 1985), this benchmark has become the de facto standard for gauging relational database performance.

The TP1 test simulates a banking system workload. The benchmark debits or credits a bank customer's account and updates related teller and branch accounts. The purpose of the benchmark is to do as many of these simultaneous debit/credit transactions as possible. Throughput is measured in Transactions Per Second (TPS). The TP1 benchmark is closely related to the Debit Credit benchmark. The chief differentiator is that the TP1 test does not require terminal emulation and support.

TP1 Test Architecture

When written in SQL, the TP1 transaction consists of three UPDATES and one INSERT. The UPDATES are for the customer's bank account balance, the corresponding teller's drawer balance, and the corresponding bank branch balance. The INSERT appends to the history table the audit trail record for that transaction.

In addition to the basic TP1 database transaction, the TP1 test suite performs the following functions:

- Generates a random branch number.
- Generates a random teller number at the selected branch.
- Generates a random account number at the selected branch for 85% of the transactions and at a different branch for 15% of the transactions.

TP1 Database Scaling

Bank accounts are actually updated rather infrequently. It takes a very large account base to generate a sustained high TPS load. For example, in the real world any banking system requiring sustained throughput of 100 transactions per second probably has millions of accounts. The I/O necessary to page such a large database in and out of memory is a significant drain on system throughput.

For this reason the standard TP1 benchmark has strict rules for sizing the branch, teller, and account tables. As TPS ratings increase, the database tables used must grow larger. This insures that audited TP1 benchmarks include a realistic amount of disk I/O, and prevents unrealistic throughput claims based on small tables that can be made memory resident and thus subject to very fast update.

The fully-cached TP1 tests presented in this paper used tables smaller than those required by the TP1 standard. This was done specifically to eliminate disk I/O operations. Early testing revealed that bus saturation occurred even with few I/O operations. Since the purpose of the initial test series was to find the cause of the bus saturation, the database tables used were sized to fit completely into system memory.

Following the fully-cached tests, a second series of TP1 tests used scaled database tables that required large amounts of disk I/O. Some of the scaled TP1 test results presented in this paper come close to the standard TP1 scaling criteria and would provide auditable test results with more tuning. As mentioned earlier the purpose here was to study the performance of the architecture for such an application rather than get an auditable TP1 number. For correct scaling of TP1 a larger database would be required which was not possible to create because of limitation of hardware resources available during this experiment.

3. RELATIONAL DATABASE ARCHITECTURE

Two-Process Architecture

The relational database system used in this investigation utilizes a client-server architecture consisting of a front-end application program and a back end database engine. The front-end handles user interfaces and compiles user input into messages containing SQL statements which are sent to the back end. The back end executes the SQL statements and returns results to the front-end, handles disk I/O, and maintains database integrity.

Because it uses two separate address spaces, the two-process architecture isolates and protects the database engine from problems in the front-end application code. It also simplifies applications development because a standard database engine can serve a wide variety of different front-end applications.
In the TP1 benchmark, the front-end process is a relatively light-weight process. This effectively emulates the client-server architecture, where the front-end process may run on personal computer or workstation and the back end on a database server.

**Single-Process Architecture Tests**

For comparison purposes, some tests were also run on an earlier version of the database that did not use the two-process architecture (single-process model).

4. **SYMMETRY SYSTEM HARDWARE ARCHITECTURE**

**Sequent Symmetry Hardware**

Sequent's Symmetry Series is a bus-based shared-memory multiprocessor (Lovett and Thakkar, 1988). A diagram is shown in Figure 1. A machine can contain from two to thirty CPUs with a peak performance of around 120 MIPS. Each processor subsystem contains a 32-bit microprocessor, a floating point unit, optional floating point accelerator, and a private cache.

The system features a 56 MB/sec pipelined system bus, up to 240 MB of main memory, and a diagnostic and console processor. Symmetry systems can support five dual-channel disk controllers (DCCs), with up to 8 disks per channel. Each channel can transfer at 1.8 MB/sec.

The DYNIX operating system is a parallel version of UNIX, designed and implemented by Sequent for their Balance and Symmetry machines. It provides all services of AT&T System V UNIX as well as Berkeley 4.2 BSD UNIX.

**Symmetry Architecture**

Since system bus and processor cache mechanisms were a major target of this investigation, a brief overview of bus and cache architectural issues is provided below:

**Caches In Uniprocessor Systems**

To sustain peak performance a uniprocessor system has to be able to access instructions and data rapidly. Main memory speeds are usually much slower than those required to support the desired access rates. Thus a cache memory that can be rapidly accessed and is transparent to software is used to sustain the processor's peak performance. The cache lies close to the processor and is a part of the memory hierarchy in the system.

Both the temporal and spatial locality of program access behavior allows a relatively small cache to satisfy a majority of processor's request. Because of this, large caches show diminishing returns in uniprocessor systems. They are relatively expensive and may not significantly increase system throughput.

**Caches In Bus-Based Shared Memory Multiprocessors**

In the bus-based shared memory multiprocessor cache memory is used to sustain peak processor performance in a manner similar to uniprocessor systems. More importantly though, caches are used to keep the processor's memory accesses off the bus.

The bus is the most critical resource in a multiprocessor system, and bus utilization in these systems is directly proportional to the cache-miss rate. For example, in a 30 CPU system if the miss-rate for individual processors goes up by 1/4%, this increase, when multiplied by 30 processors, amounts to 7 1/2% for the system. Thus bus utilization increases by 7 1/2%.

If an application exhibits a miss-rate exceeding a critical percentage, the system bus will be saturated by main memory accesses. Increases in miss-rates may be generated by legitimate application behavior or by applications operating in a non-optimal manner for the architecture. Thus there is potential to increase the performance of these later applications by realizing this behavior and fixing it.

In bus-based shared-memory multiprocessor systems, larger caches are generally worth the cost because they eliminate potential bus traffic due to cache misses.

**Symmetry Cache Coherency Protocol**

The Symmetry multiprocessor system supports an efficient copyback cache coherency protocol. This protocol enables Symmetry to scale to a larger number of processor for parallel and multi-user workloads (Thakkar, 1989). The scaling enabled by this protocol compares well with the protocols simulated by Archibald (Archibald and Baer, 1986).
The protocol allows a dirty copy of a data block to exist in one of the caches in the system. The dirty block gets copied back to memory implicitly when another cache makes a non-exclusive read access to that block. Alternatively, it gets copied back to memory explicitly when that line in the cache is replaced to make room for another block.

The Symmetry copyback cache coherency protocol (Lovett and Thakkar, 1988) supports four cache states: invalid, private, shared and modified. Both private and modified are exclusive states. The private state is a read exclusive state and modified state is a write exclusive state. The coherence protocol is similar to Illinois protocol (Papmarcos and Patel, 1984). The differences are described in (Lovett and Thakkar, 1988).

The cache coherency protocol is based on the concept of ownership. That is, to perform a write operation on a block (assuming a cache miss), a cache has to first perform an exclusive read operation on the bus to gain ownership of the block. Only then can the block be updated in the cache.

If another cache is holding the block in modified state, it has to respond to the read exclusive request and invalidate its copy. The responding cache asserts the owned line on the bus, indicating that memory should not respond to that request.

For a non-exclusive read request on the bus, all caches that hold the block in shared state will assert the shared line on the bus. The memory responds and the block is loaded into the requesting cache as shared.

Synchronization Mechanism

The synchronization mechanism on the Symmetry model uses cache-based locks. The locks are also ownership based. That is, the cache controller treats a locked read from a processor like a write operation.

Assuming a cache miss, the cache controller performs an exclusive read operation on the bus to gain ownership of the block. The atomic lock operation is then completed in the cache. These locks are optimized for multi-user systems where locks are lightly contested and the critical sections are short. They do not work well in some parallel applications where a lock is heavily contested. Several other software synchronization schemes can be used to reduce contention for the locks in the hardware (Archibald and Baer, 1986, Graunke and Thakkar, 1990).

Response latency

In general, caches in multiprocessor systems serve two masters, the processor and the bus. A cache has to respond to bus requests when it owns a dirty block, and also to processor requests. The memory only responds to a single processor access at a time, hence it can respond much faster. On Symmetry, the Sequent System Bus can be thought of as a pipe to memory. The pipe depth is optimized for memory latency. The caches have an asynchronous interface to the bus and have to support processor accesses. Thus a cache-to-cache transfer is slower than a memory-to-cache transfer in such a system.

The Sequent System Bus is an unpended (split-transaction) bus. A fixed number of requests are allowed on the bus, and responses to requests are strictly ordered. Responses to earlier requests have to come back before responses to later requests can be allowed on the bus.

The number of requests allowed on the bus is optimized for the number of cycles required by a memory response, because memory responds to the majority of bus requests. Cache responses, having longer latency, require more bus cycles than memory responses. The additional bus cycles spent waiting for non-optimal, slower-than-memory responses are wasteful of bus bandwidth because they prevent further requests from being put on the bus. These additional cycles can be classified as "hold" cycles. Thus if a cache responds to a bus request, potentially useful bus cycles are wasted as hold cycles.

Process Migration Effect On Shared-Memory Bus Traffic

In a multi-user environment, a process is scheduled to run on a processor until it gets context switched as a result of operating system scheduling (i.e., time-slicing) or when it blocks on a resource. This is known as involuntary context switch.

At present, DYNIX makes little effort to schedule a process to be allocated to the same processor after it has been context switched. Thus, a process may start on a different processor when DYNIX schedules it to run again. This is called process migration, since a process migrates from one processor to another.
The potential problem is that the process’s context may still be in its previous processor’s cache. The context then has to be moved from the previous cache to the current cache. (This only applies to dirty blocks.) This cache-to-cache traffic is totally unnecessary, and has a detrimental effect on bus and processor performance, because cache-to-cache traffic adds hold cycles on the bus.

In Symmetry Model B systems, caches are relatively small (64K bytes), and the process migration caused by involuntary context switches is not normally a significant problem. Contexts, in general, stay in the cache for a longer time in multiprocessor systems than in uniprocessor machines, and in a small cache they almost certainly wipe out previous context. Since multiprocessors provide more individual processor resources, the competing processes do not interfere as much on a per processor basis. As caches grow larger though, a bigger percentage of previous contexts may stay in the cache. Then process migration will cause cache-to-cache bus traffic. Thus, process migration can be a performance limiter in multiprocessor systems with large caches.

5. PERFORMANCE EVALUATION

Goal and methods

The goal of this investigation was to analyze the behavior of the shared-memory architecture for OLTP applications. First, fully-cached databases were used to determine the upper bound on performance of this application on Symmetry Model B hardware and software. Next, tests requiring I/O operations were used, but these tests were not fully scaled according to strict TP1 criteria because the application features that were studied did not require it.

System Configuration

A Symmetry Model B system with 16 MHz Intel 80386/80387 and Weitek 1167 processor subsystems and 64-KB local caches was used for the experiments. A system with faster processors and larger caches was also used for some experiments. 96 MB of physical memory were available in the system. About 15% of the physical memory is used by the operating system for code, internal tables, and file system I/O buffer space.

The experimental system was equipped with 12 CPU boards (24 CPUs) 3 DCCs, and 12 disks, 2 disks on each channel. Two of the disks were used for DYNIX file systems. In order to measure only comparable numbers, we used only the other four DCCs for the experiments reported here. These 10 disks were opened in raw mode, i.e., DYNIX did not provide buffering, read-ahead, or write-behind for these disks.

The number of database front-end/back end process pairs used was always equal to one less than the number of processors. One processor has to be left free in the DYNIX environment to handle operating system functions.

Results Summary

Analysis shows that processor migration can be detrimental to performance of OLTP applications because of the amount of bus traffic generated by this activity. Disabling process migration with the processor affinity function gives a significant throughput improvement for this application.

When bus traffic is reduced to allow maximum performance, the system potentially can become I/O bound. The number of I/O accesses per second is the limiting factor.

Increasing the size of system memory improved performance because caching more of the database in main memory reduced the number of I/O requests.

Early Approaches

Larger Cache Sizing Was Tried

When we first encountered the performance roll-off, we suspected that processor cache size was the problem. If processor caches were too small, the processor might be going to memory frequently to fetch data. These requests could saturate the bus. The observed cache miss rate was high, providing possible evidence of a cache that was too small. This led to testing of a processor subsystem with a cache twice the size of the Model B. This processor subsystem also ran at a 25% faster clock rate.

The larger cache did reduce bus utilization. However, system performance only improved by about 20%. As in the Model B tests, the system bus became saturated and throughput was flat after 16 CPUs, despite the bigger cache. The relatively uniform 20% performance increase seen was slightly less than the increase in clock rate.
High-Contention Locks Were Suspected

It was then obvious that the high bus traffic was caused by some other activity in the system. Bus measurements showed that a significant portion of bus cycles were being wasted as hold cycles.

The type of bus traffic that has the most potential to generate hold cycles is cache-to-cache traffic. High contention locking activity can generate large amounts of cache-to-cache traffic involving lock data structures (Graunke and Thakkar, 1990).

A particular database-related lock was suspected, and a test was run with a modified synchronization lock algorithm, designed to reduce contention. The new algorithm had no effect.

Process Migration Was Identified

Next, the Sequent Database and System Performance personnel tried to reduce the high cache miss-rate and identify the source of the high bus utilization. It turned out that the process migration problem feared in large cache systems had materialized here.

One of characteristics observed during this test was that the context switching rate for the application was too high. It was significantly higher than that observed for other multi-user environments. The majority of context switches were voluntary context switches. This rate was proportional to the message passing between front-end and back end processes in the database.

High context-switch activity generated high cache-to-cache traffic because, since the processes had voluntarily context switched to pass messages, the contexts of the switched processes were still warm in the cache.

This context then gets moved from one cache to another as the operating system schedules the processes on other processors. The cumulative system activity resulting from these moves causes the performance degradation. The degradation manifests itself as bus saturation when a significant percentage of the bus is consumed by the cache-to-cache traffic and its side effects.

Affinity Was Used To Reduce Context Switching

The way to increase performance transparently to the application is to eliminate the cache-to-cache traffic. This traffic can be eliminated by restricting the movement of processes when they are context switched. Thus we needed to override the existing process scheduling algorithm actions.

The solution was to assign a process to a processor using the DYNIX processor affinity function. The affinity function allows a process to stay on the same processor forever. Thus after a context switch the process gets restarted on the same processor. Since the cache is still warm with respect to that process, the strategy maintains processor access rate and reduces bus utilization.

This affinity function is not an ideal environment for multi-user system since this disturbs the load-balancing. Ideally we would like to use a time heuristic affinity function which does not allow a process to resume on the same processor if a certain amount of time has elapsed. However, for the purpose of this experiment the use of the available function was sufficient since we were running experiments to determine effects of using the processor affinity versus not using it. The amount of users was held constant per processor. The tests were conducted on an isolated machine thus no other load was present on the system.

Message-Passing Overhead Was Reduced

At the same time, the database developer took the initiative to develop a version of the database that passed fewer messages between the front-end and back end processes, to reduce the high context-switch rate.

This is a technique that is used in decoupled front-end/back end environments, where the application front-end and the database back end run on different machines. In such applications, network overhead caused by interprocess communication between the front-end and back end is quite significant. This overhead can be reduced by batching adjacent back end database requests into a single message from the front-end to the back end.

For example, the TP1 transaction consists of four adjacent SQL statements, three UPDATES and an INSERT. Normally this would result in four send/receive message pairs. But if the
requests are batched, only one send/receive message is required for the four commands.

This same batching facility also helps performance in a co-resident environment like the one created for the Symmetry TP1 tests. The batching technique reduced message traffic between the front-end and the back end, which reduced the context switch rate. Reduced context switching, in turn, reduces much of the bus overhead caused by process migration. It also reduces CPU-related message processing overhead caused by loading and unloading data into message buffers, verifying it, etc.

An interesting point to note is that with a fully cached database reduction of message passing between the front-end and back end achieved 91% of the throughput that was achieved by using the affinity function. This is with the number of processes being twice the number of processors.

So, the high context switch rate caused by high message passing activity did contribute significantly to the performance degradation. However, the best performance numbers are achieved by both reducing the number of messages (i.e., the number of voluntary context switches), and by stopping process migration (i.e., eliminating unnecessary cache traffic).

Performance Analysis

Observations On Two-Process Model versus Single-Process Model

When tests were run on a version of the database that did not use the two-process architecture (single-process model), some interesting comparisons could be seen. These tests led to discovering the process migration problem.

In the two-process model the context switch rate was directly proportional to the message passing rate. Because of the volume of messages passing between the front-end and back end processes, the context switch rate with the two-process model was nearly 10x the context switch rate of the single task model. Further, the two-process model generated more bus hold cycles than the single-process model.

The high context switch rate with the two-process model caused a high cache miss-rate. However, the context switch rate did not go up when processors were added. It remained constant. And, as processors were added, the miss-rate came down. This effect can be seen in the performance graphs that follow.

Results From Processor Subsystem With 2x Larger Cache And 25% Faster Clock

Tests were also run on a processor subsystem with a 2x larger cache and 25% faster clock rate, to investigate the effects of larger cache size.

The context switch rate with the larger cache/faster processor was slightly higher than for the Model B subsystem. However, fewer hold cycles were generated than with Model B, and bus utilization was better. Some bus bandwidth was still available.

Single-process tests were also run on the larger cache/faster processor subsystem.

Throughput increases with the larger cache/faster processor were lower for the two-process model than for the single-process model. Increases with the two-process model varied from 35% with 12 processors to 20% with 24 processors. This compares with improvements of 42% with 12 processors to 50% with 24 processors the for single task model. These results show that using more processors is not resulting in a throughput increase for the two-process model of the database over the single-process model. The study described in this paper was conducted because we did not see a this gain in throughput.

The larger cache/faster processor reduced the cache miss-rate, but the reductions were greater with the single-process model. For the two-process model, the larger cache/faster processor showed a miss-rate decrease of 19%. For the single-process model the decrease was 45%.

Results of performance tuning on Model B

The rest of the report describes the process of effectively tuning the database, using the two-process model, to run efficiently on Symmetry multiprocessor system.

In the attached performance graphs, Figures 2 - 5 show total TPS numbers, cache miss rate, and bus utilization for fully-cached tests on Symmetry Model B system.

Figures 6 - 9 show the same parameters with scaled database tests, which required I/O operations in addition to log writes.
Figures 10 - 11 show the effect of affinity on the demand for I/O operations, and the effect of page cache size on performance.

**Fully Cached Tests**

**Figure 2. Model B - TPS rate (affinity vs. non-affinity)**

The non-affinity test shows little increase in TPS rate as the number of processors is increased. The affinity test shows over 40% increase in TPS rate as processors are doubled.

**Figure 3. Model B - Miss rate (affinity vs. non-affinity)**

In the non-affinity case the miss-rate drops steadily as processors are added. This is because the total cache space in the system is increased, and the work done is spread across the entire system. The affinity case shows a negligible decline in the miss-rate as processors are added. This is because a fixed amount of work is assigned to each processor.

**Figure 4. Model B - Bus Utilization (affinity vs. non-affinity)**

The bus utilization in the non-affinity case starts to drop as a result of lower miss-rate. However, the bus utilization rises steadily in the affinity case because the miss-rate per processor is constant as processors are added.

**Figure 5. Model B - Hold Cycles (affinity vs. non-affinity)**

The hold cycles in the affinity case are half that of the non-affinity case. The decrease in the hold cycles for the affinity case is entirely due to stopping the process migration and decreasing message passing between the front and back ends of the database application.

**I/O Based Scaled Tests**

**Figure 6. Model B - TPS (affinity vs. non-affinity)**

In tests requiring significant disk I/O activity, the flattening of the TPS rate as processors are added is more dramatic for the affinity based test than for the non-affinity test. This flattening is entirely due to the I/O limitation of this particular configuration. The non-affinity test shows less degradation because the major limitation in this case is the hold cycles caused by process migration.

**Figure 7. Model B - Miss Rate (affinity vs. non-affinity)**

The miss-rate for the affinity test with I/O follows the miss-rate for the fully cached affinity tests. It decreases gently as more processors are added. It is much lower than in fully-cached affinity test because much the I/O limitation allows less work to be done. The miss-rate for the non-affinity test is missing because of an error in recording the results. However it is little different from the non-affinity fully-cached test.

**Figure 8. Model B - Bus Utilization (affinity vs non-affinity)**

The bus utilization for the affinity case follows the miss-rate trend. The bus utilization with affinity is much lower than for the non-affinity case. This is really because the affinity test is more I/O bound.

**Figure 9. Model B - Hold Cycles (affinity vs. non-affinity)**

The affinity case shows less than 3% hold cycles. Once again, this indicates that the test is I/O bound. The hold cycles in the non-affinity case with I/O are marginally lower than the fully-cached non-affinity case. This indicates that the non-affinity case is only marginally I/O bound. (Note that the bus utilization and cache miss-rate are similar indications for the non-affinity case.)

**Figure 10. Model B - I/O Performance (affinity vs. non-affinity)**

Six of the 10 disks were accessed heavily during the tests. This partitioning was done in accordance with TP1 benchmark specifications. The other four disks were lightly accessed. The peak rate for accessing 2K bytes blocks randomly from each of the heavily-loaded disks is around 36 I/O's per second. This gives an aggregate rate of 210 I/O's per second for these disks. The graphs show that Model B with affinity approaches this number with 207 I/O's per second, indicating that the application is now I/O bound.
The affinity test results indicate that as the page cache size (M bytes) is increased the performance of the test approaches that of the fully cached affinity test. The test results shown are for 24 processors.

6. CONCLUSIONS

The TPI testing revealed a hierarchy of performance bottlenecks which show the limitations of the current hardware and software. Each type of test highlighted a different type of limitation. These limitations were overcome by tuning the application to utilize the shared memory architecture more effectively. The hardware monitoring tools showed the effects the initial effects of process migration on the hardware and was later to effectively as tuning aid.

The fully-cached testing was only limited initially by bus bandwidth. After the addition of affinity and message batching, the fully-cached tests showed linear increases in performance throughout the processor range, with no bottlenecks at all in the tested range.

The scaled testing revealed a bottleneck caused by disk I/O. These tests show that limiting factor for the OLTP application is I/O throughput and not processor speed. The I/O throughput can be improved by adding more disks to the system or by having large memory sub-system that can cache the large portions of database.

Process migration was shown to cause significant performance degradation for this application. A simple recommendation for an operating system process scheduler would be to use a time-base scheduling heuristic algorithm. This would resolve the problem caused by process migration. A process would be scheduled to stay on a processor and would be reinvoked on the same processor if the time duration from the last context switch was less than some number.

Shared memory architectures have been shown to be suitable for relational database OLTP operations, and can deliver very high performance when the system and application are properly tuned. There is a need for an I/O solution that will match the performance of the next generation of microprocessors, which will operate in the 100-plus MIPS range. This I/O need is even higher for multiprocessor system, which will incorporate several of these microprocessors.

Acknowledgements

We would like to thank Prof. Dan Siewiorek and other reviewers for improving the readability of the paper. We would also like to thank Betty Fuller for assistance in putting this paper together. Finally, though not the least, we would like to thank Joe Turner for editorial assistance.

References


Database TP1 Tests: Model-B Non-All. VS All.
Two Process Model - With I/O

Figure 9

Figure 10

Figure 11

Figure 12

Figure 13

Figure 14

Figure 15

Figure 16

Figure 17