PAPER *Special Issue on Parallel/Distributed Computing and Networking*

DCLUE: A Distributed Cluster Emulator*

Krishna KANT[†], **Amit SAHOO**[††], *and* **Nrupal JANI**[†], *Nonmembers*

SUMMARY Given the availability of high-speed Ethernet and HW based protocol offload, clustered systems using a commodity network fabric (e.g., TCP/IP over Ethernet) are expected to become more attractive for a range of e-business and data center applications. In this paper, we describe a comprehensive simulation tool to study the performance of clustered database systems using such a fabric. The simulation tool currently supports both TCP and SCTP as the transport protocol and models an Oracle 9i like clustered DBMS running a TPC-C like workload. The model can be used to study a wide variety of issues regarding the performance of clustered DBMS systems including the impact of enhancements to network layers (transport, IP, MAC), QoS mechanisms or latency improvements, and cluster-wide power control issues.

key words: *Unified Ethernet Fabric, Clustered systems, performance modeling, database emulation, distributed caching.*

1. Introduction

In the e-business environment, mid-tier and backend applications have traditionally been implemented on SMPs (symmetric multiprocessors) because of their easier programming model and efficient inter-process communication. However, SMP implementations have several serious drawbacks including high cost and inability to grow the system gradually as the need arises. Thus, there is a definite trend towards cluster based computing even for well entrenched SMP applications such as large databases. This trend is further aided by the emergence of high bandwidth, low-latency cluster interconnect technologies such as Infiniband architecture (IBA) and HW offloaded TCP/IP over Ethernet [2], [4]. On the software side, there are already solutions available for running applications that do no require a painstaking manual partitioning in order to minimize inter-process communication (IPC). However, there isn't much information available in the open literature on the performance, scalability and stress behavior of *commercial* clustering solutions. In this paper we describe a database cluster simulation tool that we have designed and provide some sample results. A detailed study of performance scalability is contained in [3].

The primary focus of our study is a cluster Ethernet as the "unified" clustering fabric. IP/Ethernet is attractive as a high performance clustering fabric for a variety of reasons: (a) commodity availability of Ethernet at 10 Gb/s or higher speeds, (b) optimized HW implementation of TCP/IP that can reduce communication latencies, (c) developments in storage over IP area which makes Ethernet a cost effective alternative to Fiber channel, and (d) large entrenched base of IP/Ethernet infrastructure. Furthermore, a single "unified pipe" coming into a server is highly desirable for high density "blade" servers where space and power are at a premium. However, such an approach requires that the unified fabric work almost as well as isolated fabrics under stress conditions. The tool developed here allows the study of such issues and development of necessary QoS mechanisms for unified fabrics.

2. Clustered Database Architecture

Clustered DBMS implementations cover a wide range in terms of the level of coupling of various nodes. On one extreme, there is the "shared nothing" approach, where each node has its own independent memory and IO subsystem. In this case, the database must necessarily be partitioned among the nodes. A more coupled approach is "shared IO" approach, where all nodes access a centralized IO subsystem which holds the database. The IO subsystem in this case is invariably a Fiber-channel based SAN. DCLUE supports both of these models. In particular, the basic model assumes a distributed iSCSI based storage available at each node. One attraction of such a "distributed storage" model is that it allows for an inexpensive IO system at each node which expands naturally with the cluster size. A centralized SAN based storage model is also supported; however, the details of the SAN are currently not being modeled.

In a clustered DBMS, each node stores the currently accessed data portions in what is known as the '*buffer cache*', and this makes a data coherence scheme essential. The are two major coherence schemes for strict data consistency, both supported by DCLUE:

1. Read/write locking (RWL): This is the traditional scheme where reading requires a shared lock but writing requires an exclusive lock. Write lock also requires *invalidation* of all existing copies. It is clear that RWL can result in significant overhead in terms of locking, invalidation and the associated messaging.

[†]Intel Corporation
[††]University of California, Davis
[*]This work has been submitted to PODC 2005

2. Multi-version concurrency control (MCC): This scheme [1] creates a new *version* of the item on each update. MCC avoids any "read-locks" since a transaction can always find the appropriate version of the data to read. Write/update accesses still require locking, however, there is no need for a traditional "invalidation"; instead, the concurrency control needs to ensure that only the most recent version is written to. The price for MCC must be paid in terms of managing multiple versions, additional memory requirements, fatter IPC data messages, and more disk IO.

The basic value proposition of distributed caching is that retrieving data out of the remote buffer cache is significantly cheaper than reading it from the disk (even in case of local disks). Thus, one would expect small clusters to perform well even with the traditional "SW TCP" solutions. With HW TCP implementations, good scaling should be possible even for rather large clusters.

The distributed caching in DCLUE is based upon a directory based scheme known as *cache fusion* [5] that is used in Oracle 9i. Cache fusion works as follows: Suppose that a node A experiences a miss on DB block X in its local buffer cache. Node A then determines (via a local table lookup) that some node, say B, holds the directory information for this block. A requests the block X from node B, which in turn directs the appropriate data holder C to send the block to A. In case the no node holds the data, A obtains block X from the disk (local or remote). Note that it is possible that $A = B$, or $B = C$; in these cases some operations become local and the corresponding messaging is not needed.

3. TPC-C like Database Workload

The TPC-C benchmark (`http://www.tpc.org/tpcc`) is a natural choice for an online transaction processing (OLTP) database workload because of its popularity and availability of detailed characterization data. TPC-C models operations of a wholesale parts supplier operating out of a number of warehouses and their associated sales districts. The workload has 5 transactions, namely *new-order, payment, order status, delivery* and *stock level* . The nominal fractions of these transactions are 43%, 43%, 5%, 5% and 4% respectively. The performance metric reported by TPC-C is the number of "new-orders" processed *per minute* and is expressed as tpm-C.

The benchmark involves 9 tables: *warehouse, district, customer, stock, item, new-order, order, order-line* and *history.* Of these, the first 5 tables are fixed, but others are variable. The benchmark is designed such that *the database size increases linearly with the throughput.* The largest tables are typically customer and stock and may require significant space for their indices.

A notable characteristic of TPC-C transactions is that they all refer to a single warehouse. This, coupled with the fact that most tables have the number of warehouses as a multiplier, makes TPC-C database trivially partitionable: assign equal blocks of warehouses to each server and direct queries based on the warehouse. For this reason, TPC-C is usually considered an inappropriate workload for clustering studies. We address this weakness by not necessarily directing queries to the right server. Instead, we introduce the notion of *affinity.* An affinity of $\alpha([0, 1])$ means that the query goes to the right server with probability α and to a random server with probability $1 - \alpha$.

4. Cluster simulation Model

The simulation model was developed using OPNET (`www.opnet.com`). DCLUE was built on top of the OPNET provided TPAL (transport adaptation layer) which can support multiple transports underneath.

The model implements distributed caching, multiversion concurrency control, row/page locking, logging, disk IO, database tables, table operations, buffering, IPC handling, application processing, scheduling, thread switching, processor-memory data transfers, etc., often in painstaking detail. As a result, it requires a rather fine-grain model calibration. This is a problem in spite of a wealth of available measurement data on TPC-C, TCP processing, iSCSI processing, etc. On the positive side however, *the model isn't dependent on high level results that would be easily invalidated by a change in system parameters.* For example, the hit ratio in the buffer cache is not an input parameter; instead, it is a result of the actual buffer cache management done by the simulation. Similarly, the number of locks acquired per transaction, IPC messages sent/received per transaction, log blocks written to the disk, blocks read from the disk, data versions created per block, context switches per transaction, etc. all fall out of the actual functioning of the simulation rather than being artificially provided as some inconsistent set of values.

In spite of the detail, DCLUE obviously could not mimic a real system at a fine grain level; the purpose of DCLUE is to merely implement the most important functionality from a performance perspective and thereby allow sensitivity studies. Some of the high-level functionality missing from DCLUE are failure recovery and checkpointing since these are not essential for our purposes. Nevertheless, given the model calibration based on actual measurement data, the results can provide valuable insights into the performance of OLTP workloads on a cluster. The model also allows a number of what if studies by changing a wide variety of parameters which could be difficult to change in a measurement setup.

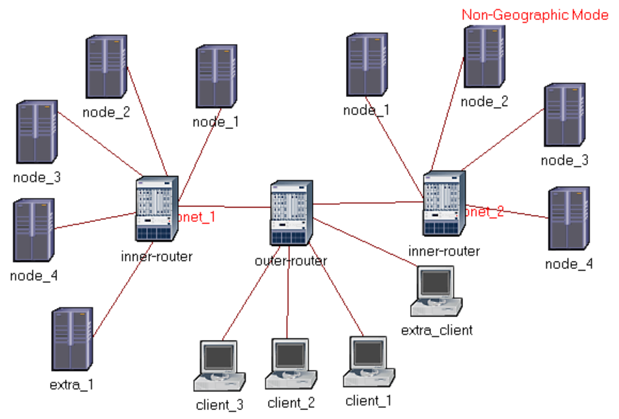

Fig. 1 A sample DCLUE model w/ 2 latas & 4 nodes per lata

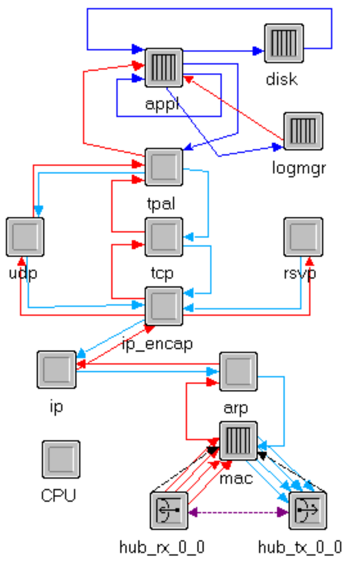

Fig. 2 Node model of DCLUE servers

Figure 1 shows the DCLUE network model. The network is organized as one or more "subclusters" which we call LATAs (borrowed from telecom). The subclusters are connected via an "outer-router" (or an "outer-switch" if we only want layer 2 switching), at which the clients also home in. Each server has internal disk subsystems for normal IO and logging, but not all of them may be used. In the distributed storage configuration, the disks are accessed remotely via the iSCSI protocol and locally via the SCSI protocol . In the centralized (or SAN based) storage configuration, the set of all IO subsystems forms a virtual SAN which is accessed via some SAN fabric.

One of the objectives for the model is to study potential ill effects of running IPC and storage traffic on the normal Ethernet network that carries miscellaneous other types of traffic. For this, the model allows some extra clients and servers("extra_client" and "extra_server" in Fig 1) to be added to the cluster. These can run some additional applications and cause that traffic to interfere with DBMS traffic on various links and routers.

During initialization, each server establishes 2 TCP connections to every other server: one for IPC messages (data & control) and the other for iSCSI related traffic (command, status, data, etc.). The reason for separate connection is to allow QoS studies that treat IPC and storage separately. By default, TCP parameters are set appropriately for a data center environment rather than the WAN and can be changed easily. The client-server TCP connections are established dynamically on a per "business transaction" basis. A business transaction consists of the sequence of TPC-C transactions starting with the *new-order* in the proportions specified in section 3.

Fig 2 shows the internals of the node model for regular servers. The modules with vertical bars show those that implement queuing, whereas others only implement the logic plus pure delays. The paths between modules are "streams" used to deliver data or control. Each module is described further via a state machine (or a "process model"), but these are not shown. The additions that we have made are the appl, disk and logmgr modules; others are standard OPNET implementations. The node model is generic and thus shows certain features that are not being used here (e.g., UDP, RSVP, IP encapsulation, etc.). The node shows only a single NIC and the corresponding transmit and receive modules. Other nodes types (e.g., extra server, client and extra client) have similar node models except for the application level details.

5. Architecture of DCLUE

5.1 Database Representation

In order to allow for detailed table operations, DCLUE builds the entire database in memory according to TPC-C rules. However, it only maintains enough information about each table row in order to correctly handle the essence of each query. It turns out that beyond the basic information (e.g., warehouse id, district id, customer id, list of items ordered, item quantities, etc.) just one or two parameters are adequate to correctly execute each query. Thus the database takes much less space than the real one. The *actual* row sizes are still used to compute such parameters as number of rows per page, rows per subpage (for subpage level locking), etc. In spite of these savings, storage requirements become excessive for large throughputs. We use a consistent scaling mechanism to address this issue (discussed later). In addition to the tables themselves, DCLUE also maintains a B^+ tree based index for each table. Although indices are normally fully cached, they are treated just like tables and may be only partially cached.

Since the entire database is sitting in the memory, buffer cache operations merely relate to status changes and list operations – the simulator does not need to perform any actual disk IO. The buffer cache is maintained separately for each table (typical in TPC-C implementations) as governed by the specified "caching fraction". The same applies to indices. The page replacement policy is simple LRU with a few modifications.

To allow for fine-grain locking, a page (or disk block) is divided up into a number of *subpages*. A subpage contains one or more *complete* rows, and the subpage size can be chosen independently for each table. This was essential since certain tables (e.g., district & new-order) have a much higher contention than others and thus benefit from small subpage sizes. A small subpage results in more overhead both in terms of simulation (more memory space and slower processing) and the simulated system (need to acquire more locks for queries working on a sequence of rows).

5.2 Locking Mechanisms

The basic locking mechanism in DCLUE is fairly simple. First, every transaction type acquires locks in a consistent order so as to prevent deadlocks. For TPC-C, the first locks are always on warehouse and/or district table. If locking is not possible, the transaction is put on a wait list, to be woken up later by a transaction that releases the contended lock. Note that the locks already acquired are not released before going into a wait state. For other tables, the inability to lock results in the release of all locks and a rollback of the transaction to the point of first lock attempt. In this case, the transaction waits for a exponentially distributed amount of time and then tries again. A transaction retries a few times, and if it doesn't succeed, it aborts. In order to avoid holding locks unnecessarily, the locking in DCLUE is actually a 2-phase process: *latching* (or intention locking) followed by *locking*.

1. In phase 1, the transaction goes through the query plan to determine what subpages it needs to operate on and which one of those need to be locked. It posts requests for missing pages to the directory node . The lock requests are also transmitted to the directory node, which acquires "latches" on the requested subpages. The difference between a latch and a lock is that latches are compatible with one another, i.e., multiple transactions can hold a latch on the same subpage. However, latch is not compatible with a lock, and any attempt to latch a locked subpage fails immediately.
2. The transaction enters phase 2 when all the requested data has already been placed in its buffer cache and all latches have been acquired. (The requested pages are pinned in the buffer cache so that they don't get replaced.) The transaction then makes a single request to the directory node to convert all its latches into locks. If the conversion is not possible, the transaction may wait or release locks/latches and do a delayed retry as explained above.

The above scheme is likely to be rudimentary compared with the ones used in commercial DBMS, but it appears to work fairly well even at low affinity values. Sophisticated schemes (e.g., multi-level locking) may yield better performance under heavy contention, but should not change many of the sensitivity results.

5.3 Multiversion Concurrency Control

MCC implementation requires mechanisms to store multiple time-stamped versions of each lockable item, which in our case is a subpage. The memory required for creating new versions is allocated in DCLUE from a special version memory pool. The pool size is specified independently for each table to ensure that small but heavily used tables such as district can have many versions without their pool space being eaten up by monster tables like customer or stock. If the version memory pool for any table is exhausted, pages are stolen from the corresponding buffer cache.[†] For large tables like customer and stock, this will always happen since we set the pool size to 0 for them.

As more and more versions are created for subpages within a block, the effective size of the block grows. If another node requests a block, all its versions are transferred, and the requestor has the job of figuring out which version it really needs. Such a scheme makes all versions available at the requesting site in one shot, but may be wasteful of memory. Although a new version of a subpage may be created as soon as a transaction (that holds lock on it) attempts to modify it, the new version will not be visible to other transactions until the transaction commits successfully and releases the lock. If the transaction rolls back, the privately created new version will simply be discarded. DCLUE separately maintains the "global time-stamp" for each subpage, which is the highest timestamp across all nodes. An update must always use this version. A minimum time-stamp is also maintained at each node so that unneeded versions can be discarded.

Each transaction itself carries a time-stamp, determined from the time the transaction successfully converts all its latches into locks. Basically, the idea is that to maintain serializability, the transaction must use subpage versions that are current as of this time. For a read-only transaction, the transaction time-stamp is the time when it starts execution. Finally, when a transaction with the smallest timestamp is retired, any subpage versions with a lower or equal timestamp can be discarded.

[†]If all pages in the buffer-cache are pinned and thus no stealing is possible, the transaction aborts.

5.4 Disk IO and Logging

DCLUE emulates disk IO in significant detail in order to capture some essential characteristics of disk IO behavior. In particular, for each table, disk reads and writes are queued separately according to the page number and an elevator scheduling algorithm is used for both, with switches between read and write cycles. Since the queues may contain duplicate requests from different nodes, all duplicate IO requests are retired immediately after the IO finishes. The latency and path-length consequences of disk IO are accounted for, and so are such situations as references to a dirty block while it is being written to disk. In this case, the disk write is still allowed to complete, but the block is not evicted from the buffer cache on disk write completion.

DCLUE implements the normal lazy disk write model where the modified pages are written back to disk only when they are evicted (and have the highest version number). Thus, a disk write might happen long after the transaction is committed. In order to ensure database consistency in case of a failure, all data modified by a transaction is written to a *separate* log disk. A transaction must wait for log data to be safely flushed to the disk before committing. Logging does not go through the normal file-caching path of the OS and thus is much less processor intensive. Log writes are also much faster at the disk because of sequential writes.

DCLUE allows for distributed logging where each node does its own private logging. This avoids a centralized bottleneck but can make recovery much more expensive. More centralized logging (i.e., per subnet or per cluster) is also supported.

5.5 Application Processing

A DCLUE client implements a number of "client threads", each of which generates business transactions at a specified rate. A business transaction, in turn, consists of a sequence of sub-transactions (or simply transactions), each of which involves a request-response interaction with the server. The server is chosen according to affinity and other load distribution characteristics and remains the same for all sub-transactions. Clients communicate with servers using *transaction* packets.

A transaction packet arriving at a server is entered in a data structure where it stays until finished (retired, aborted, timed-out, or rejected outright due to lack of resources). The inter-process communication is accomplished via packets of a different type called *messages*. A message is smaller and shares some crucial fields (e.g., trans_id) with the transaction packets. All packets involve their own processing (TCP/IP processing plus some application IPC layer processing) in addition to triggering application processing.

Application processing proceeds according to the query plan for each transaction type. The total overhead of completing a transaction is the accumulation of the overheads of individual steps such as disk IO, buffer cache management, locking/unlocking, versioning, logging, TCP/IP processing, context switching, table operations, data sorting, transaction commit, etc. This requires a very detailed parameterization, and the data for this was pulled from many sources. In particular, a lot of TPC-C processing data is taken and scaled from the NASA report [6] and current TPC-C measurements. The data related to cache, bus and memory channel performance is taken from recent measurements and well-reviewed TPC-C projections within Intel. The data concerning network stack processing overheads is taken from authors' work and other ongoing work in Intel on acceleration of TPC, iSCSI and other protocols [2], [4], [8], [9].

5.6 Thread Management

Each accepted transaction is assigned an *application thread* which stays with it until done. This thread is responsible for all application processing and may experience many context switches. In particular, the thread blocks voluntarily for each receive and disk read. The thread may also be switched out involuntarily to schedule *system threads* for such tasks as TCP/IP processing, disk IO handling, logging, locking/latching, and unlocking. The simulation attempts to implement a network IO scheme similar to real systems. In particular, a receive operation interrupts application processing to schedule TCP/IP processing (if done in SW) and for placing data in the user buffer. The typical situation of application processing culminating in a message *send* is handled by ensuring that the application execution path-length is fully simulated before the send becomes "eligible".

Thread management and scheduling incurs some cost, which is included as an input parameter. The cost of a context switch is modeled as follows. There are three costs associated with a context switch:

1. Basic cost: CPU cycles needed to save the state of the running thread and restoring that of the new thread. This part is typically quite small and is denoted as $S_{\min}$.
2. Working set accumulation stalls: CPU stalls (in cycles) while the lost portion of the working set is rebuilt (via memory-cache transfers of required cache-lines).
3. Additional bus/memory traffic: The rebuilding of the working set increases load on processor bus and memory channels which results in additional memory access latencies and hence CPU stalls for other memory accesses as well.

In order to model these with some degree of realism, we exploited detailed measurement data for Redhat Linux 7.3 OS [7]. The measurements list the context switch cycles as a function of working set size and number of competing threads. Based on these measurements, we find that the following equation matches the measurements quite remarkably. Let η denote the fraction of processor cache occupied from average working set size considerations. That is,

$$\eta = [N_{app}S_{app} + N_{sys}S_{sys}]/cache\_size \tag{1}$$

where N_x and S_x denote the number of active threads of type x and their working set sizes. Then, assuming a fairly homogenous system, the fraction of working set lost between two successive schedulings of the same thread, denoted $L(\eta)$, can be approximated by the following equation:

$$L(\eta) = \begin{cases} L_{\min} + (1 - L_{\min})(1 - 2^{-2\eta+1}) & \text{if } \eta \geq 0.5 \\ 4L_{\min}\eta^2 & \text{if } \eta \leq 0.5 \end{cases} \tag{2}$$

where $L_{\min}$ is the fraction of working set lost when $\eta = 0.5$. $L_{\min}$ was consistently found to be around 0.1. Now, the CPU stall (in CPU cycles) caused by the switching in of thread type x is given by:

$$S_x = S_{\min} + B_x L(\eta) \tag{3}$$

where B_x is the total cost of building the entire working set for thread type x (expressed in CPU cycles). It can be estimated from the working set size, average burst size and memory pipeline delay (including post-L2, address bus, data bus and memory channel delays).

¿From other TPC-C specific data, we know that the average TPC-C working set size is around 32KB. We did not have reliable information about system threads and assumed a working set size of 4KB.

5.7 Bus and Memory Modeling

All processing overheads in DCLUE are specified in terms of path-length (i.e., average number of instructions required to accomplish a task). In order to convert these to CPU time (or cycles) we also need estimates of *cycles per instruction* (CPI). The overall CPI depends not only on the basic processor architecture but also on CPU stalls caused by the memory subsystem. In fact, the overall CPI can be expressed as the base CPI (i.e., CPI under infinite L2 cache) plus a component contributed by CPU stalls due to latencies in data retrieval from the memory. In particular,

$$CPI_{tot} = CPI_{base} + BF \times MPI \times mem\_acc\_latency \tag{4}$$

where BF (blocking factor) is the fraction of memory access latency that is visible to the CPU and MPI is average *misses per instruction.* The BF value depends on the platform, and the established TPC-C BF value for the modeled platform (0.78) was used. The MPI depends on the cache size and detailed caching behavior of various components of the workload. The memory access latency obviously depends on the queuing delays in the address & data busses and in the memory channels. Other than direct IO, the traffic hitting the bus and memory are misses out of the cache which means that MPI and memory access latency are not independent.

In order to estimate CPI_{base}, we modify the baseline CPI obtained from unclustered TPC-C measurements based on the increased working set resulting from affinity less than 1. Basically, the idea is that as the affinity decreases, the amount of data to be maintained expands proportionately. Similarly, starting with the MPI for an unclustered system, we do some modifications according to a power law of the form: a doubling of working set increases MPI by X%, where X is a parameter. The parameter X is obtained from measurements. Estimating memory access latency itself can be quite involved as it includes 4 key components: (a) latency of actually putting out the request on the address bus, which we call post-l2 latency, (b) address bus latency, (c) memory channel latency, and (d) data bus latency. While address and data busses can be modeled as simple queues with deterministic service times, the memory channel is modeled as a queuing station (with number of non-lock step channels as the number of servers) in series with some pure delay. With this, the queuing delays and hence the overall memory access latencies can be computed analytically.

6. Cluster Performance Modeling Studies

DCLUE takes a large number of input parameters which are specified via 3 different mechanisms: (a) input file, (b) header file, and (c) direct parameter setting via GUI. The basic business transactions are generated according to the specified distributions. The generation supports a 2-phase semi-Markov arrival process. In addition, it is possible to generate traffic with cyclic overload pattern. In this section we use the DCLUE model to obtain a number of interesting results on scalability and the effects of contention. As stated earlier, although standard TPC-C specification is exploited heavily in the implementation and model calibration, we are interested in scenarios beyond basic TPC-C particularly in terms of the role of IPC in clustered databases. Running DCLUE produces detailed statistics for each client and server node. The server statistics are also aggregated on a per-lata basis. This feature is useful if the traffic distribution across latas or nodes within a lata is made uneven in order to study impact of focused load imbalances. Both client and server statistics are also averaged globally.

The base model calibration was done for Intel Pentium IV class dual-processor (DP) servers for which unclustered TPC-C measurements and validated platform performance models were readily available. In particular, the baseline server configuration is a 3.2GHz P4

DP system with 1 MB second level cache, 133 MHz bus and 16GB of DDR-266 memory. One such node delivers about 50K (unclustered) tpm-C performance, which amounts to a database with about 4K warehouses.

Unfortunately, even a small cluster of such nodes will require very long simulation time and huge amounts of memory. The need for $>$ 4GB memory which would require the complexity of reworking the simulation to use PSE/AWE on 32-bit machines. To avoid this problem, we consistently scaled all relevant parameters by a factor of 100x. This includes CPU frequency, processor bus, memory channels, PCI bus, disks, links, routers etc. Furthermore, all CPU overheads are expressed as "path-lengths" (i.e., number of instructions required to accomplish the operation) or as path-length equivalents. As for the database itself, a slow-down in all platform and OS parameters will automatically reduce the throughput (and hence the number of warehouses) by 100x. With the above scaling, it is possible to simulate reasonable sized clusters. The results can then be scaled back to get an estimate of the performance of the original system.

Detailed results from DCLUE are discussed in [3] and will not be discussed in full detail here. Instead, we only include some sample scalability results in order to provide some idea of the nature of results that DCLUE can provide. In particular, Fig 3 shows cluster throughput vs. cluster size and affinity as a parameter. The affinity 1.0 case is shown just as a reference and corresponds to the case of perfect scaling. As expected, the scaling gets progressively poorer as the affinity rises. However, the interesting part is an almost linear scaling from 2 or 3 nodes to 10 nodes. For larger clusters, locking related and topological issues start to come into effect.

At high affinities, the reason for continued scaling is the lack of any shared bottlenecks in the system. In fact, most resources increase linearly with the cluster size. For example, each new node adds not just CPUs, but also memory, memory channels, processor bus, normal and logging disks, and router links. If the network grows by adding more subnets, the stress on each inner-router also remains unchanged. Even the lock contention per page stays the same since TPC-C mandates that the database size increase linearly with the throughput. For low affinities, the low realized throughput prevents the bus from becoming a bottleneck for moderate cluster sizes, despite a significant increase in the MPI.

Fig 4 shows the impact of slower growth of DB size as a function of throughput. For this we assumed that for up to 90K tpm-C, the database sizing is according to TPC-C rules. However, beyond this, the growth rate of warehouses grows as square root of the additional throughput, rather than linearly. With this, the contention for the data increases as the cluster size increases. Consequently, the throughput no longer goes up linearly with the cluster size. Figures 5 and 6 show two other situations of limited resources. In Fig 5, we assume that the forwarding rate of the routers is reduced from the normal 10000 packets/sec to 4000 packets/sec. This causes the throughput to saturate beyond 8 connected servers. Fig 6 shows a scenario where one node in the cluster is responsible for all logging operations. This *centralized logging* makes recovery much easier than the logging-per-node assumed elsewhere. However the price is paid in terms of poorer scalability due to a centralized bottleneck.

Figures 7 and 8 show two other interesting results from the model. Fig. 7 shows the average number of versions created by the multiversion concurrency control mechanism as a function of affinity and the number of nodes. The version count is computed only over the sub-pages that are actually touched – not over all database subpages. The fact that lower affinity leads to fewer versions is completely opposite to what we had originally expected. The reason for smaller versions is the much higher lock contention which keeps a lid on the number of versions in existence. The decrease with number of nodes also looks unexpected, and is related to faster eviction of the page from the buffer cache. Finally, Fig. 8 shows throughput as a function of number of configured threads. As expected, the throughput rises with number of threads up to a point and then flattens out. The more interesting result – not shown here – is that with large number of threads, the achieved throughput becomes quite sensitive to overload . It is found that under overload, excess threads only result in the processor cache to thrash and hence result in poorer performance.

These cases plus a number of others reported in [3] show the value of DCLUE in studying clustered DBMS performance under a variety of scenarios relating to available resources, traffic distributions, implementation alternatives (e.g., HW vs. SW TCP), QoS configurations, communication latencies, sizing parameters, etc. Because of very detailed model calibration, it is also possible to move significantly away from standard TPC-C characteristics. In particular, it is possible to study how the latency sensitivity of the workload varies as computation to communication ratio is varied, or the fraction of light-weight vs. heavy weight queries is varied. It should, however, be kept in mind that as the characteristics move significantly away from standard TPC-C the fidelity of the model could deteriorate in terms of issues like computation of MPI and bus/memory traffic.

7. Conclusions

In this paper, we have described a comprehensive model of clustered database systems. The model allows a wide variety of studies with clustered DBMS systems and thus provides a valuable vehicle for understanding

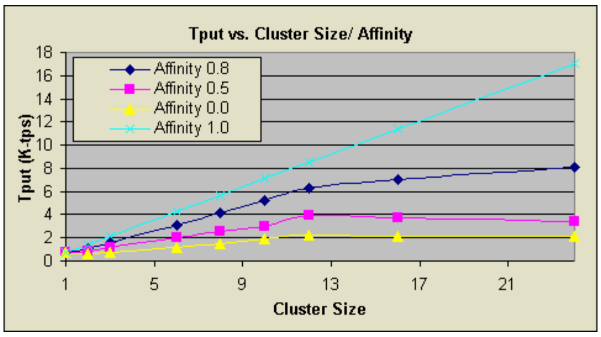

Fig. 3 Scaling vs. nodes and affinity

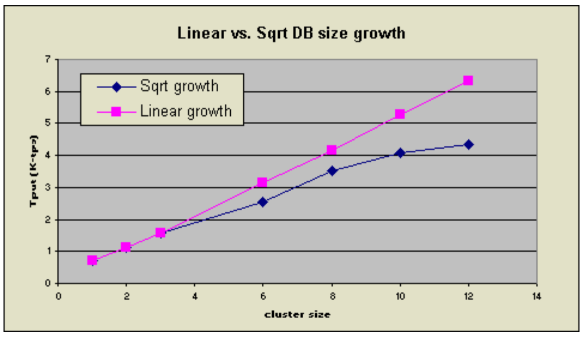

Fig. 4 Impact of slower growth in DB size

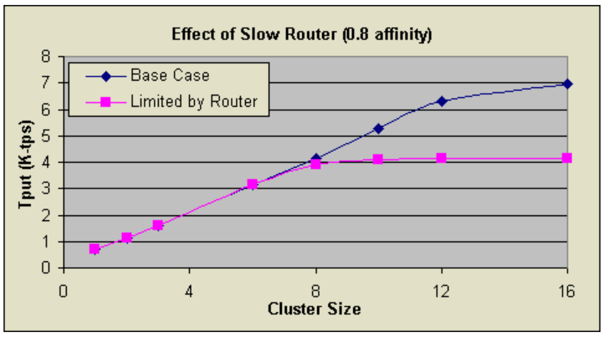

Fig. 5 Impact of router forwarding rate on scalability

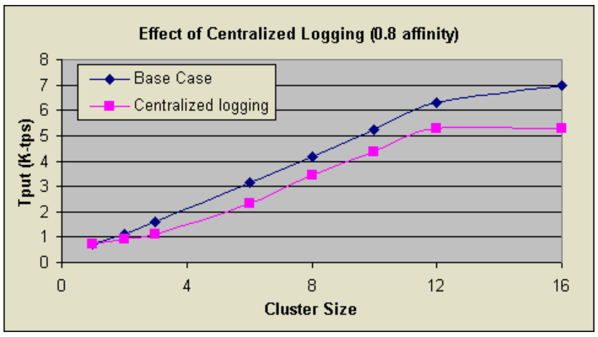

Fig. 6 Impact of single node logging on scalability

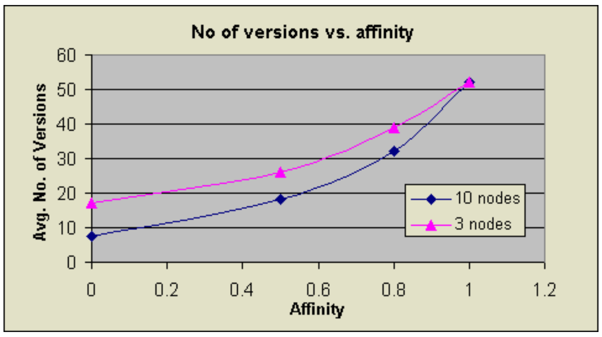

Fig. 7 No of subpage versions vs. affinity

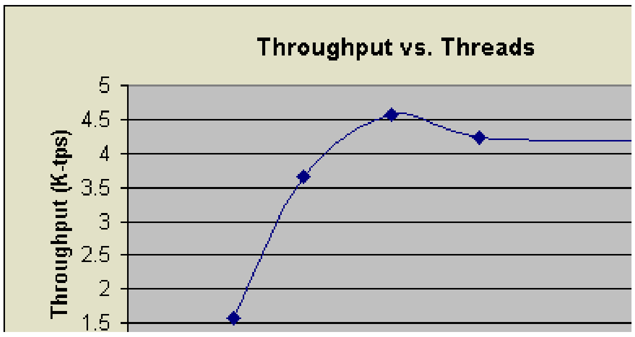

Fig. 8 Throughput vs. no of threads

their performance. The model is particularly valuable in examining the impact of fabric characteristics on the application latency. In particular, application level impact of any new developments at MAC, IP and transport layer can be readily examined using DCLUE. For example, the ongoing IEEE work on MAC level congestion control schemes can be examined using DCLUE. Other uses include study of cluster wide power control studies, study of advantages of transport layer multicasting with traditional RW locking, study of utility computing paradigm (e.g., virtual clusters), etc.

References

[1] P.A. Bernstein, and N. Goodman, "Multiversion concurrency control — theory and algorithms", ACM Trans. on Database Systems, vol. 8, no. 4, pp.465-483, Dec. 1983.
[2] G. Regnier et al., "TCP onloading for data center servers", Special issue of IEEE Computer on Internet data centers, Nov 2004 (Eds. K. Kant & P. Mohapatra).
[3] K. Kant, and A. Sahoo, "Clustered DBMS Scalability under Unified Ethernet Fabric", available at kkant.ccwebhost.com/DCLUE.
[4] K. Kant, "TCP offload performance for front-end servers", Proc. GLOBECOM, San Francisco, CA, Dec. 2003.
[5] T. Lahiri et al., "Cache Fusion: Extending shared disk clusters with shared caches", Proc. 27th VLDB conference, Rome, Italy, 2001.
[6] S. Leutenegger, and D. Dias, "A modeling study of the TPC-C benchmark", ACM SIGMOD Record archive, vol. 22, no. 2, pp.22-31, June 1993.
[7] P. Deng, "Telecom Linux performance evaluation", Intel measurement and evaluation report, Aug. 2002.
[8] A. Joglekar, "iSCSI Technology Investigation", Intel measurement and evaluation report, Nov. 2004.
[9] S.R. King and F.L. Berry, "Software RDMA over TCP/IP on a general purpose CPU", submitted for publication.