DCLUE: A Distributed Cluster Emulator

Krishna Kant
Intel Corporation

Amit Sahoo
Univ. of California, Davis

Nrupal Jani
Intel Corporation

Abstract

Given the availability of high-speed Ethernet and HW based protocol offload, clustered systems using the commodity 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 model 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.

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 some serious drawbacks including high cost and inability to grow the system gradually as the need arises. In particular, the cost of a SMP system increases very non-linearly with the number of ways (or processors). Furthermore, as the processor and interconnect speeds go up, the NUMAness becomes an increasingly difficult issue to deal with. Thus, if the application can be ported to a clustered system without significant changes and yet achieve a good performance scaling, the "scale-out" model forms a compelling alternative. This is further aided by the emergence of high bandwidth, low-latency cluster interconnect technologies such as Infiniband architecture (IBA) [15] and HW offloaded TCP/IP over Ethernet [10, 7]. On the software side, there are already solutions available for running applications without a painstaking manual partitioning in order to minimize inter-process communication (IPC). For example in the database space, clustered systems such as Oracle 9i/10g have claimed that a good concurrency control model coupled with a distributed caching service can avoid the need for database partitioning [12]. However, there isn't much information available in the open literature on the performance, scalability and stress behavior of such systems. In this paper we provide an analysis of scalability and performance of such clusters running database applications.

The primary focus of our study is a cluster using TCP/IP over Ethernet as the "unified" clustering fabric. In spite of the existence of several specialized clustering fabrics such as IBA, Myrinet, QSNet, etc, IP/Ethernet is expected to become attractive at high end 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 almost to the level of specialized fabrics, (c) developments in storage over IP area which makes Ethernet a cost effective alternative to the much more expensive Fiber channel based storage technologies, and (d) large entrenched base of IP/Ethernet infrastructure (including switches, routers, etc.) which specialized technologies cannot match. 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.[1] 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 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 nodes – either statically or dynamically. 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 (system area network). 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 inexpensive IO system at each node which expands naturally with the cluster

[1] Availability considerations will still dictate redundant standby or load-balanced interfaces; however, that aspect doesn't change the argument.

size. A centralized SAN based storage model is also supported; however, the details of the SAN are currently not being modeled.

Database systems invariably use a significant portion of the available memory to cache the data portions currently being worked on. This is usually called the '*buffer cache*. In a clustered DBMS, each node provides its own buffer cache which 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. That is, each node can read its local copy so long as it is not write-locked, whereas acquiring a write lock requires that no copy has any lock on it. Write lock also requires *invalidation* of all existing copies so that those copies will not be used following the release of the write lock. It is clear that RWL can result in significant overhead in terms of locking, invalidation and the associated messaging.

2. Multi-version concurrency control (MCC): This scheme [2] 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 (due to less efficient memory use).

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, the overhead and end-to-end latency of IPC vs. that of disk IO are crucial parameters for the performance and scalability of cache fusion based clustered DBMS. It is well known that the traditional OS Kernel based TCP/IP implementations are quite inefficient [10]. Nevertheless, the corresponding IPC overheads and latencies are still considerably smaller than those for disk IO. 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.

Oracle 9i style distributed caching uses a directory based scheme that proceeds 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. The sequence of actions is then as follows:

1. A requests the block X from node B. B looks up the directory and returns positive or negative response to A.

2. In case of a negative response, A obtains block X from the disk (local or remote).

3. In case of a positive response, A waits to receive the block from some node C which is determined by B as the data supplier. B sends a message to C, and C responds to A directly with the block. (The last one is IPC data message, all others are control messages).

4. A eventually informs B of successful retrieval so that B can update the directory indicating A also as the data holder. (If A had to evict a block from its buffer to accommodate the new one, it informs B of that too.)

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.

The IPC data transfer is not limited to a single DB page – the transfer size could range anywhere from 4KB to 64KB. The optimal transfer size depends on a number of factors, and we do not attempt to adjust it for different runs of the model. Instead, we assume a basic IPC transfer size of 8 KB (same as disk block size).

Other than the block transfer and directory management related IPC traffic, the scheme involves a number of other IPC messages for such things as write lock acquisition/release, transaction commits/aborts, notification of block caching/evictions to the directory node, checkpointing, directory migration, etc. These operations may result in a significant number of additional IPC messages between nodes.

3 TPC-C like Database Workload

Given its popularity and availability of detailed characterization data, the TPC-C benchmark (`http://www.tpc.org/tpcc/default.asp`) is a natural choice for an OLTP database workload. TPC-C models operations of a wholesale parts supplier operating out of a number of warehouses and their associated sales districts. Each warehouse supplies 10 sales districts, and each district serves 3000 customers. The database manages 100K parts in terms of orders, prices, stock level, etc. The workload has 5 transactions, namely new-order (enter a new order which requests 10 parts on the average), payment, order status, delivery (process a batch of ten orders for delivery), and stock level (level of stock of the items ordered by last 20 orders). The nominal fractions of these transactions are 43%, 43%, 5%, 5% and 4% respectively.[2] The performance metric

[2] Actually, TPC-C allows new-orders to go up to 44% at the cost of delivery, but this may have some undesirable consequences.

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. New-order table grows & shrinks as new orders come in and are retired. The last 4 tables keep a permanent record of transactional operations and thus only grow. The benchmark is designed such that *the database size increases linearly with the throughput*. In particular, the number of configured warehouses is approximately tpm-C/12.5. Sizes of all tables, except item, are multiples of warehouses. The item table stays constant at 100K rows. The largest tables are typically customer and stock and may require significant space for their indices. Although the variable tables like order, order-line and history are also quite large, access to them is quite localized.

A notable characteristic of TPC-C transactions is that they all refer to a single warehouse. In fact, according to the specification, a given "terminal" always generates transactions with the same warehouse-id. This, coupled with the fact that most tables have #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. (TPC-C does mandate some remote operations however; in particular, 1% of the new-orders are served out of a remote warehouse. However, given the small percentage and the fact that "remote" here doesn't necessarily mean another node, unmodified TPC-C workload is not very interesting from clustering perspective).

We address TPC-C's weakness by tweaking the workload behavior according to our needs. In particular, we still partition the database in blocks of warehouses, but do not necessarily direct queries to the right sever. Instead, we introduce the notion of *affinity*. An affinity of 1.0 corresponds to the case where a query always goes to the server that hosts the referenced warehouse. An affinity of $\alpha < 1$ 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`). OPNET provides a fairly complete emulation of the network infrastructure including TCP, IP, and Ethernet MAC layers, QoS support, commercial switches and routers, etc. 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. This allows us complete freedom in choosing the cached fractions of various tables and their indices. 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.

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 iSCSI protocol and via SCSI protocol locally. 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. The SAN fabric is implicitly assumed to have low latency and adequate resources and is not explicitly modeled.

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 to be added to the cluster (distinguished in the model by a different address range). These clients/servers can run some additional applications and cause that traffic to interfere with DBMS traffic on various links and routers. For exam-

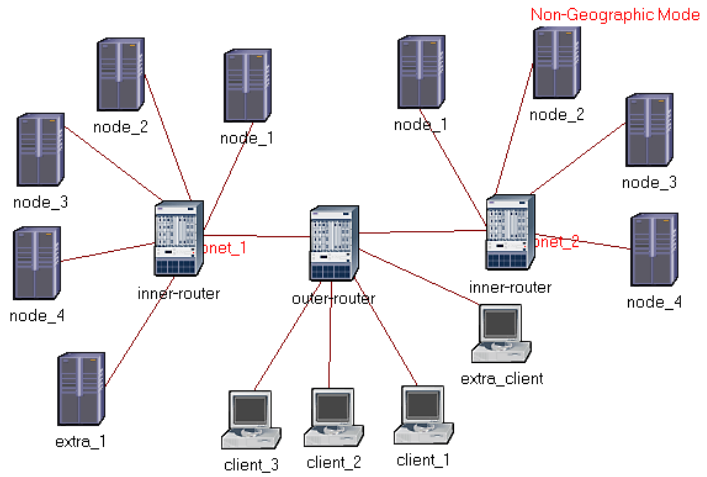

Fig 1: A sample DCLUE model w/ 2 latas & 4 nodes per lata

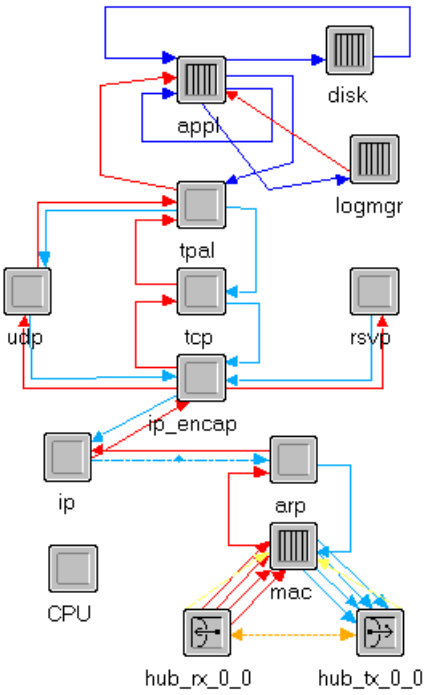

Fig 2: Node model of DCLUE servers

ple, Fig 1 shows the nodes marked “extra_client” and “extra_server” whose traffic interferes with regular DBMS traffic on inter-lata links.

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 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.

The base DCLUE model is built on top of OPNET supplied TPAL (transport adaptation layer). Fig 2 shows the internals of the node model for regular servers. According to OPNET GUI conventions, 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., USDP, 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. For example, the client includes not only an application processing module but also a traffic generation and control module.

As shown in Fig 2, OPNET currently supports only TCP and UDP, of which only TCP is in use. Using UDP instead is trivial from TPAL; however, the application currently has no provision to deal with lost packets. It is possible to add other reliable transports such as SCTP underneath TPAL and be able to use them. Unfortunately, OPNET currently does not supply standard SCTP modules. We do have a SCTP implementation of the data transfer part, which means that it assumes that the connection parameters have been set up appropriately on both ends of the intended connection. The implementation is ad hoc and not designed to effortlessly slide under TPAL. Nevertheless, it is possible to study DCLUE performance under SCTP. Some potential uses of this are to ex-

amine the impact of features that TCP doesn't provide, e.g., multi-streaming, multi-homing and reliable datagram service.

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 corectly handle the essence of each query. Thus, there is no need to allocate space for every field. 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 parmeters 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. The order table also needs an auxiliary index for the "max-select" operation (i.e., select most recent order of a given customer), but such an index is not implemented currently.

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, which are normally fully cached. The caching policy is simple LRU except that a tail search is required for each replacement to ensure that pinned pages are not replaced.

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). Given subpage level locking, it is most convenient to implement version control and logging also at the subpage level.

DCLUE recognizes 11 different states for a subpage. The state of the subpage changes as a result of latching/locking, query excution and buffer cache operations. The states are:

unlk_clean_in, unlk_dirty_in, latch_clean_in, latch_dirty_in, lock_clean_in, lock_dirty_in, unlk_stale, dkw_unused, latch_stale, unlk_out, latch_out

Here the in/out distinction refers to whether the subpage (and, in fact, the entire page) is in buffer cache or not. The clean/dirty setting is done during query execution depending on the operation (read, update, insert, delete). The locking related state is indicated by unlocked, latched or locked (see below for details). "dkw_unused" means that a dirty page being written out to disk (prior to eviction) and no transaction has attempted to use it (i.e., eviction can indeed happen when the disk write completes). Any attempted usage will change the state appropriately but the disk write will still be allowed to proceed and no eviction will happen. The "stale" state refers to the situation where the subpage is still allocated in the buffer cache but contains invalid data. Maintaining states correctly in the face of transaction retries, aborts, timeouts, and orphaned IPC messages turned out to be the most challenging part in the simulator.

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 avoid deadlocks. That is, more aggressive schemes that allow deadlocks to occur and then break them are not used in DCLUE. 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 partially acquired locks are not released before going into wait. 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. (The client may still decide to retry it after getting the unsuccessful response.) 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. Since some of the pages may not be resident in the buffer cache, it posts requests for them to the directory node (which follows the normal RAC procedure described in section 2. 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 place 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.

Given the vast literature on database locking (and on concurrency control mechanisms in general) [17], 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 metadata about the subpage my further indicate what portions of the subpage actually differ from the previous version. We assume 40 bytes for time-stamp plus other metadata per 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 high use but small 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.[3] 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 (original page plus space from the overflow area) 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. Alternatively, shipping only the needed version is more efficient memory-wise but may require repeated version transfers.

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 be simply 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 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, and it should read versions current as of this time. Finally, when a transaction with the smallest timestamp is retired, any subpage versions with a lower or equal timestamp can be discarded.

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 transanction 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.

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

5.5 Application Processing

The clients implement 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 individual 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. The clients also perform a number of other functions, but those are not relevant to current discussion. A transaction packet has 38 fields including 9 list fields to keep track of such things as items ordered, order-ids, item prices, etc.

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 (only 20 fields) and shares some crucial fields (e.g., trans_id) with the transaction packets. Messages are created and destroyed as needed. Messages involve their own processing (TCP/IP processing plus some application IPC layer processing) in addition to trigering 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 (required for stock-level transaction), transaction commit, etc. This requires a very detailed parameterization, and the data for this was pulled from many sources (too numerous to describe them here). In particular, a lot of TPC-C processing data is taken and scaled from the NASA report [18] 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 author's work and other ongoing work in Intel on acceleration of TPC, iSCSI and other protocols [10, 7, 21, 20]. Admittedly, there are still "guestimates" of several parametes; however, those should be adequate for the purposes of understanding the general nature of clustered DBMS performance.

5.6 Thread Mangement

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 messaging related TCP/IP processing, disk IO handling, logging, locking/latching, and unlocking. The TPC-C system is dominated by application threads, however. 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 send is handled by ensuring the following:

(a) The application execution path-length is fully simulated before the send becomes "eligible". Note that the CPU overhead of application processing must be simulated explicitly (it doesn't just happen automatically as a result of executing the application logic!).

(b) The TCP/IP processing related to an "eligible" send takes precedence over application processing for any transaction, but still has lower priority than receives.[4]

Thread management and scheduling incurs some cost, which is included as an input parameter. The number of context switches per transaction falls out of the scheduling scheme described above. However, we also need to accurately model the cost of context switches. 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 [19]. 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,

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

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

[4] This could cause receive livelock situation as in real systems, but we have not addressed this issue further.

two successive schedulings of the same thread, denoted $L(\eta)$, can be approximated by the following equation:

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

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 swithing in of thread type x is given by:

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

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 procesor 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 latencies in data retrieval from the memory. In particular,

$$CPI_{tot} = CPI_{base} + BF \times MPI \times mem\_access\_latency \quad (4)$$

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.

Although the above equation is reasonable on an overall basis, individual processing components could use very different types of instructions or have very different caching behavior. In general, it is not possible to estimate context specific CPI except in some very specific circumstances. For example, CRC-32 calculation (used for iSCSI or RDMA) has a very low CPI which may be known. In all other cases, we continue to use overall CPI for converting instructions to CPU cycles; however, this approach is merely an approximation.

The overheads associated with task switching, locking and chipset communication are expressed as "equivalent" path-lengths as well, even though these are perhaps closer to being pure CPU cycles. Since much of the model calibration is based on the data obtained from unclustered TPC-C measurements, the corresponding *baseline CPI* is the relevant one for converting these to CPU cycles. We allow a slight modification of this when considering clustered systems 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. Based on unclustered TPC-C measurements that quantify the impact of increased working set on MPI, we modify the MPI according to a power law. This power law is of the following form: a doubling of working set increases MPI by X%, where X is a parameter. The modified MPI value, say MPI', is not the overall MPI that the simulation will provide. Instead, it is an intermediate quantity that is used to estimate how the increase in computational path-length increases the traffic on the bus. The simulation accumulates statistics on overall path-length PL_{tot}, transaction throughput λ_{tot} and CPU utilization U_{tot}. Let F denote the processor frequency. Then,

$$\lambda_{tot} = (F \times U_{tot}/CPI_{tot})/PLtot \quad (5)$$

Thus CPI_{tot} can be estimated. Since memory access latency is also available from the simulation, eqn (4) could then be used to estimate overall MPI, if required.

Estimating memory access latency itself can be quite involved. To begin with let us consider the latency of actually retrieving data from memory upon a cache miss. This includes 4 key components: (a) latency of actually putting out request on the 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 itself involves several stages. Following the normal practice, we model memory channel as a queuing station (with number of non-lock step channels as the number of servers) in series with some pure delay. Although the 3 queuing stations involved here could have been simulated, we instead used simple analytic modeling of these [8]. Thus, the key parameter required for modeling is the memory access traffic and corresponding traffic on address, data and memory busses. The memory reference traffic includes:

1. Memory references as a result of computational path-length. These are estimated using a simple multiplier for the current estimate of the path-length. This multiplier is estimated from available measurements and can be adjusted as better measurements become available.

2. Memory references involved in activities such as task

switching or lock spinning (accumulated separately from regular path-length since these mostly affect CPI).

3. Memory references due to memory-to-memory copies associated with IO.

4. Memory references due to IO data transfer (disk, iSCSI, IPC, and client-server traffic) but excluding IO related memory to memory copies.

Note that all IO data transfer involves access to memory channel and address bus (for snooping); however, only a portion of this may be actually brought into the cache and used. This fraction is another parameter and it determines the data bus use. The are other differences between ABUS, DBUS and memory usage as well. In particular, memory references must be reduced by the HITM fraction, for which we assume a constant value. Also, ABUS gets additional traffic due to invalidations required for exclusive access to shared data. This is yet another parameter that is assumed to be a constant. Finally, both DBUS and DDR memory channels involve "bubbles" which depend on the details of the access pattern. For simplicity, these too are are assumed to be constant fractions. The bus/memory modeling can be enhanced if necessary, however, we find that in most instances, bus/memory were not the bottleneck and hence a simple model is perhaps adequate. In particular, busses are generally not a bottleneck for standard unclustered TPC-C setups. This remains so in cluster setup as well since the increase in memory references is matched by path-length increase as well. Furthermore, as the affinity increases, the throughput decreases rather rapidly; therefore, the significant increase in IPC traffic does not result in busses being a bottleneck.

The inclusion of bus/memory model introduces a circularity in the model. In particular, the estimation of MPI' quantity is obviously dependent on the current throughput, and the current throughput and the current estimate of MPI' affects bus/memory traffic and hence the achieved throughput. As expected, this is handled via an iterative estimation procedure. This can cause stability problems in simulations. To see this, note that the purpose of the simulation is to find the offered load that satisfies all of the following requirements:

1. The bottleneck resource (e.g., CPU) is utilized as much as possible.

2. Nearly all of the offered load translates into the *carried load*.

3. The carried load is maximized. This typically means that no resource should be overcommitted.

Since we do not know a priori what offered load to use, multiple simulation runs are required based on "guessed" values of offered load. It is found that when the offered load is close to the desired value, the throughput might vary significantly over successive periods and thus throw the iterative process off track. In particular, it was invariably found that during the warmup period, there is generally more stress on resources as each node attempts to accumulate the right kind of data in its buffer cache. The behavior becomes more balanced as the steady state approaches. Unfortunately, there are a number of other reasons for instabilities even during the steady state.

1. TPC-C transactions vary substantially in their resource requirements. For example, the stock_level transaction works on up to 200 orders and delivery modifies information about 20 orders. At the same time, the fraction of these transactions is small. Thus, the behavior could depend significantly on how many of these transactions happen to be running.

2. Since IPC data messages are about 35x larger than control messages, the current mix of data requests vs others (e.g., lock acquisitions & releases) can affect performance significantly. Similarly, disk IO being much slower than IPC, the current mix of IPC vs. storage traffic can affect things significantly.

3. There is a wide variation in the CPU usage by various activities involved in transaction processing. For example, both transaction initiation and transaction commit path-lengths are large, and so is the path-length of max-select operation. This variation is directly reflected in the simulation since the scheduling of processing happens at a rather coarse grain.

4. The lock acquisition failure reinforces instability. For example, if the transactions at a certain node are currently experiencing a lot of lock failures, these transactions will go into the wait list or go away for a timed retry. This almost guarantees a low contention period followed by another high contention period.

In order to dampen the impact of these factors, we found it necessary to use exponential smoothing in a few places. In particular, we found that instead of computing MPI' based on offered or carried loads, it was better to compute it based on an "effective load" which takes both offered and carried loads into account. Basically, the idea of this estimation is to increase effective load when we seem to be making progress (i.e., able to carry more load) and backoff when not. Although ad hoc, the scheme does appear to work satisfactorily.

It is perhaps clear from this discussion that obtaining the "optimal offered load" for a given configuration could be a very time-consuming trial and error process, especially in situations where the performance is not being limited by CPU saturation. This results in some variability and inconsistency in the results as some of the cases in this paper show.

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 parameters that may desired to be changed most often are listed in appendix A and are given via the input file. The appendix also briefly discusses the nature of other parameters but does not actually list them.

One aspect of DCLUE not discussed so far is in the area of traffic generation and control. The basic business transactions are generated according to the specified distributions. The generation supports a 2-phase semi-Markov arrival process. The residence time in each phase, arrival rate in each phase and phase time distribution can be specified. In addition, it is possible to generate scenarios where the load with cycle overload. One of the parameters given in the input file is "processor congestion threshold", which is used to limit congestion at the server. DCLUE defines 3 processing priority levels, numbered 0..2. Priority 0 (lowest) is for transactions, priority 2 for important messages (generally those that release resources) and priority 1 for rest of the messages. The congestion threshold specifies abatement, onset and discard threshold for each of the 3 priority levels. For convenience, all thresholds are specified in units of seconds, but then converted into the number of transactions/messages by using the specified arrival rates. For example, the discard threshold for transactions is currently set to 24 seconds; therefore, if the system has already accumulated 24 seconds worth of unprocessed or partially processed transactions, any new arrivals are simply discarded and a "transaction reject" response is sent to the client. The client can, if it so desires, retry the transaction with some delay. The objective is to provide feeback rate control at the client, but this is currently not implemented.

In this section we use the DCLUE model to obtain a number of interesting results on scalability, latency sensitivity and congestion conditions. 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 lata's or nodes within a lata is made uneven in order to study impact of focussed 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 about 4K warehouse database.

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 means, for example, (1) Ethernet network model is 10baseT instead of 1000baseAE, (2) disk parameters (seek, rotation, data transfer) are slowed down by a factor of 100, (3) CPU, bus and memory channel frequencies are cut down to 32MHz, 1.33MHz, and 1.33MHz respectively, and (4) Various other delays such as chipset, IP packet forwarding, context switch, interrupt handling, etc. are also increased by a factor of 100x. In order to allow for a convenient scaling of all processing overheads, all input parameters are expressed as "path-lengths" (i.e., number of instructions required to accomplish the operation) or as or path-length equivalents. This ensures that a speed cut of CPU by 100x automatically scales everything by 100x. Finally, as for the database itself, a slowdown in all platform and OS parameters will automatically reduce the throughput (and hence the number of warehouses) by 100x – the only scaling required is for the item table, which does not depend on number of warehouses. This is done by reducing the number of items from 100K to 1000.

With the above scaling, it is possible to simulate reasonable sized clusters. The results must be scaled back to correspond to the original system.

Detailed results from DCLUE are discussed in [9] 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 issues start to come into effect. Also, topological issues also come into play. For very small (i.e., 2 or 3 node clusters), the behavior can also be different, and becomes more pronounced with lower affinity.

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 many others. These are: 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

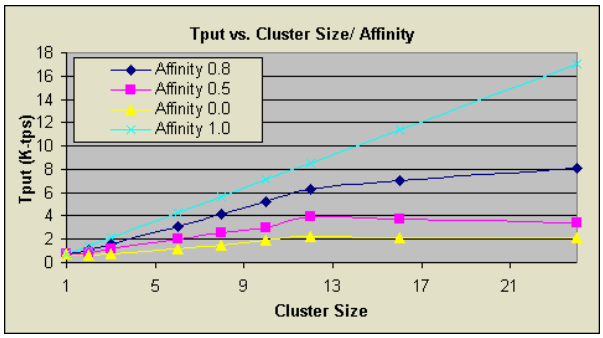

Fig 3: Scaling vs. nodes and affinity

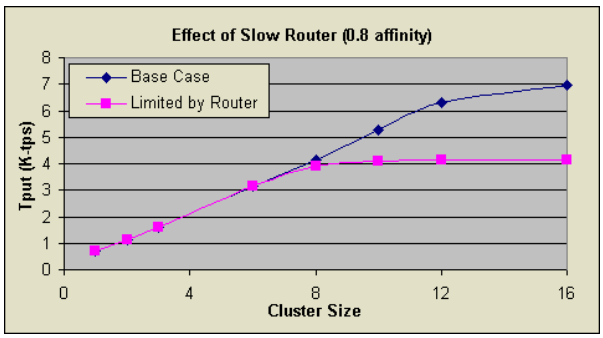

Fig 4: Impact of router forwarding rate on scalability

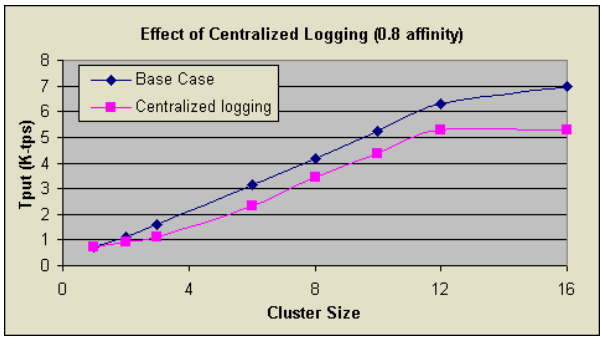

Fig 5: Impact of single node logging on scalability

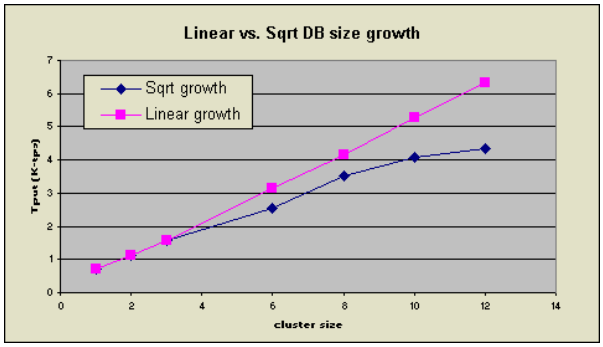

Fig 6: Impact of slower growth in DB size

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. At low affinity values, although the MPI grows significantly, the low realized throughput in this case prevents bus from becoming a bottleneck for moderate cluster sizes. The remaining figures show cases where shared bottlenecks are introduced artificially resulting in poorer scalability. Fig 4 shows a case where 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 5 shows another scenario, one where a single node is responsible for all logging operations instead of each node performing its own logging. Centralized logging makes recovery easier but at the cost of potential bottleneck during normal operation. It is seen that the performance in this case is consistently lower. Finally, Fig 6 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 (No of warehouses calculated assuming 12.5 tpm-C per warehouse). However, beyond this, the growth rate of warehouses goes a 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.

These cases plus a number of others reported in [9] 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. For example, fractions of various query types can be changed, and so can various path lengths, switching overheads, etc. 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 charcteristics 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 their performance. The model is particularly valuable in examining the impact of fabric characteristics on the application latency. In par-

ticular, 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. We plan to use DCLUE extensively for a variety of studies concerning transport layer enhancements for the data center.

References

[1] B. Benton, "Infiniband's superiority over Myrinet and QsNet for high performance computing", whitepaper at www.FabricNetworks.com.

[2] P.A. Bernstein and N. Goodman, "Multiversion concurrency control — theory and algorithms", ACM Trans on Database Systems ., 8(4):465–483, December 1983.

[3] D. Culler, R. Karp, et. al., "LogP: Towards a realistic model of parallel computation", Proc. of 4th ACM sigplan symp. on principles and practice of parallel programming, San Diego, CA, May 1993.

[4] R. Dimitrov and A. Skjellum, "Impact of latency on application performance", Proc. of 4th MPI developer & user conference, Ithaca, NY, March 2000.

[5] D. Dunning, G. Regnier, et. al., "The virtual interface architecture - a protected, zero-copy user-level interface to networks", IEEE Micro, March 1998, pp66-76.

[6] K. Kant, *Introduction to computer system performance modelling*, McGraw Hill, 1992.

[7] G. Regnier, S. Makineni, et. al., "TCP onloading for data center servers", Special issue of IEEE Computer on Internet data centers, Nov 2004 (Eds. K. Kant & P. Mohapatra).

[8] K. Kant, "An Evaluation of Memory Compression Alternatives", Proc. of CAECW (Computer Architecture Evaluation using Commercial Workloads), Feb 2003, Anaheim, CA.

[9] K. Kant and A. Sahoo, "Clustered DBMS Scalability under Unified Ethernet Fabric", available at kkant.ccwebhost.com/DCLUE.

[10] K. Kant, "TCP offload performance for front-end servers", Proc. of GLOBECOM 2003, Dec 2003, San Francisco, CA.

[11] A.C. Klaiber and H.M. Levy, "A comparison of message passing and shared memory architecture for data parallel programs", ISCA 1994, pp 94-105.

[12] T. Lahiri, V. Srihari, et. al., "Cach Fusion: Extending shared disk clusters with shared caches", Proc. 27th VLDB conference, Rome, Italy 2001.

[13] J. Liedtke, K. Elphinstone, et. al., "Achieved IPC performance", Proc. of 6th workshop on hot topics in operating systems, May 1997, Chatham, MA.

[14] J. Pinkerton, "The case for RDMA", available at `www.rdmaconsortium.org`

[15] T. Shanley, *Infiniband Network Architecture*, Mindshare Inc., 2002.

[16] T. Shanley, *The unabridged Pentium 4*, Mindshare Inc., 2004.

[17] E. Rahm, "Concurrency and coherency control in database sharing systems", Technical Report 1993, Institut fr Informatik, Leipzig Germany

[18] S. Leutenegger and D. Dias, "A modeling study of the TPC-C benchmark", ACM SIGMOD Record archive, Volume 22 , Issue 2 (June 1993), pp22 - 31

[19] P. Deng, "Telecom Linux performance evaluation", Intel measurement and evaluation report, Aug 2002.

[20] A. Joglekar, "iSCSI Technology Investigation", Intel measurement and evaluation report, Nov 2004.

[21] S.R. King and F.L. Berry, "Software RDMA over TCP/IP on a general purpose CPU", submitted for publication.

A Sample Input File for DCLUE

The table below lists the important parameters that can be set directly from the DCLUE input file. In addition, there are about 50 other parameters that were specified via #defines in a single header file called "cluster.h" and are not included here. These include, for example, bus/memory modeling parameters, TPC-C table sizing parameters, message sizing parameters, etc. It is important to note that all of these parameters include the 100x scaling that model assumes implicitly. This means that all timing parameters should really be interpreted as 1/100th of given values and all rate parameters should be interpreted as 100x of given values.

The table below also does not include most of the parameters specified by OPNET provided network stack, but those can also be set freely as desired. For example, nearly all of the TCP parameters, IP parameters, MAC parameters, link parameters, etc. are not included here and need to be set directly in the relevant models.

simulation reset time	100.0 #secs
debug level	”555-06”
debug start time	”12000 -1”
debug end time	”1000 -1”
whouse-server affinity	1.0 # Prob of going to right server
measurement interval	10.0 # block size in secs
max active threads	512
multiversion conc control	TRUE # false =¿normal conc cntrl
max trans retries at server	3 # This CANNOT exceed 9
max trans retries at client	0 # Independent of server retries
distributed storage	TRUE # false =¿ SAN storage
frac of page locked	”0.5 0.05 0.1 0.1 0.2 0.2 0.2 0.2 0.1 0.2 -1”
trans timeout period	”100.0, 60.0, 60.0, 300.0, 300.0 -1”
min think time/thread	0.0 # Min delay after trans finishes
client threads	1000 # Use 100x of new order trans rate
relative arr rate in state 1	1.0 # Ratio of arrival rate in state 1 vs 0
arr state 0 duration	5000.0
arr state 1 duration	5000.0
phase duration dist	”exponential”
phase duration parm2	0.0 # 0.0 ok for exponential/constant
overload start time	10000.0
overload magnitude	5.0
overload duration	40.0 # Should be less than cycle time
overload cycle time	90.0 #This is a repeating cycle
avg lock retry time	3.0 # use x/2 + unif(0,x.2ñ) for nth retry
avg trans retry time	10.0 # where x is the avg time given
numbe of disks	2 # Number of regular disks
number of log disks	2 # Number of log disks
max disk xfr rate (MB/sec)	1.5 # scaled down by factor of 100
min disk xfr rate (MB/sec)	0.3 # scaled down by factor of 100
min disk pure delay	2.0e-1 # in secs/block, 100x scaling
max disk pure delay	16.0e-1 # in secs/block, 100x scaling
avg disk cache span	16 # Must be an even number
disk read/write elevator cycles	1 # Read cycles per write cycle
table storage fractions	”1.0 1.0 0.05 0.15 1.0 0.1 0.05 0.025 0.05 1.0 1.0 0.5 0.5 1.0 -1”
table storage frac for versions	”0.1 20.0 0.0 0.0 0.0 0.1 0.1 0.1 0.1 0.0 0.0 0.0 0.0 0.0 2.0 -1”
client server delay	0.3 # Addl delay between client & server
trans type fractions	”0.43 0.43 0.05 0.05 0.04 -1”
traffic dist between LATAs	” -1” # just -1 means uniform distribution
traffic dist between servers	” -1” # just -1 means uniform distribution
global cong control timer	2.0 # in secs
cong ignore period	500.0 # T_ignore timer (Huge for now)
traffic throttle period	2.0 #T_throttle timer
proc cong thresholds	”12.0 15.0 24.0 54.0 60.0 75.0 86.0 91.0 99.0 -1 1.0”
proc cong throttling frac	”0.40 0.70 1.0 -1”

Table 1: Runtime DCLUE parameters

client CPU speed (MHz)	32 # 3.2GHz UP, scaled down by 100
client CPU CPI	8.0 # Overall CPI
server CPU speed (MHz)	32 # 3.2GHz ST, converted to HT!
server CPU CPI	5.4 # Core CPI w/ HT on
conn setup PL frac	0.75 # setup vs. teardown breakup
client trans prep PL frac	0.50 # trans prep vs. resp proc breakup
chipset pure delay (PL equiv)	5000 # delays incl MCH, IOH, PCI, etc.
chipset stall time (PL equiv)	333 # DMA setup, IO bus xfr, MCH
lock stall time (PL equiv)	2000 # Excludes thread switch
thread switch time (PL equiv)	2000 # Thread switching time
TCP stall time (PL equiv)	2000 #* 200 # Kernel locks & context switch
Interrupt service time (PL equiv)	4000 #* 400
conn setup/teardown PL	138800 #* 13880
disk IO initiation PL	2500 # Per disk block
disk IO completion PL	3000 # Excl. buf-cache & file system oprns
iSCSI IO init PL	1000 #* 500 Same on both src & target sides
iSCSI IO completion PL	1200 #* 500 Same on both src & target sides
CRC per KB PL	2333 #* 0
client trans prep/resp PL	250000
base msg send PL	1600 #* 400 # TCP & RDMA for 64B msg
base msg recv PL	2000 #* 500 # TCP & RDMA for 64B msg
msg send PL/KB	3000 #* 200 # Msg send/recv PL per KB
msg recv PL/KB	3000 #* 300 # Msg send/recv PL per KB
thread mgmnt PL	1500 # Thrd mgmnt (excl switching)
base appl msg PL	500 # Appl level msg prepartion
trans initiation PL	35000
directory lookup PL	3500 # Excludes locking
directory update PL	2000 # Evict & data notifications
buf cache ins/del PL	2500 # For insert/delete separately
lock management PL	2500 # Lock acquire/release PL
lock wait mgmnt PL	1000 # Lock wait/release PL
commit processing PL	26500 # Commit proc (excl lock rel, logging)
resp preparation PL	5000 # Trans resp formatting PL
table operation PL	10000 # Random select & updates
sequential select PL	1000 # Sequential select after first one
index operation PL	2500 # Direct select using an index
non-unique select PL	25000 # PL for non-unique selects
invalidation PL	2000 # Invalidation PL
list sort PL	200 # For sort only after table join
misc appl code PL	”4700 800 1300 26100 300 -1” #per trans
file system oprn PL	2000 # FS oprns (excl buf-cache)
version management PL	2500 # Version management
trans retry mgmnt PL	2000 # trans retry mgmnt PL
client_1.new order trans rate	12.5 # Per sec (not per min)
client_2.new order trans rate	12.5 # Per sec (not per min)
client_3.new order trans rate	12.5 # Per sec (not per min)

Table 2: Runtime DCLUE parameters: continued