Performance and Scalability of Client–Server Database Architectures

Alexios Delis and **Nick Roussopoulos**[*]
Computer Science Department
University of Maryland
College Park, MD 20742

Abstract

Recent developments in software and hardware changed the way database systems are built and operate. In this paper we present database architectures based on the Client–Server paradigm and study their performance and scalability under different query/update workloads. The architectures are: Standard Client–Server, Client–Server with Multiple Disks, and Enhanced Client–Server. Data replication and client query result caching are used as the main mechanisms to improve the query throughput. The role of the server is to maintain system–wide data consistency and in the case of Enhanced Client–Server to selectively propagate updates on demand. Our study shows that except for the case of mostly update workloads, the Standard Client–Server architecture is outperformed by the other two architectures by one or more orders of magnitude. The Client–Server with Multiple Disks architecture offers performance comparable to that achieved by the Enhanced Client–Server for up to 100 clients, but the latter scales up a lot better for higher number of clients.

1 Introduction

Until recently, high throughput database processing was undertaken by a large mainframe that was the dedicated machine for all the data processing [ABGM90]. The quest for even higher performance led to the investigation of multiprocessor systems in databases and database machines during the past decade [DGK$^+$86, BS88, BE89]. These efforts focused predominantly on the optimization of large join operations by using multiple disks and processors. Their major disadvantage though was their excessive cost. Aside from their limited commercial success they managed to achieve some of their objectives [BE89].

During the last decade we have experienced a number of developments that are changing the way Database Management Systems (DBMSs) are build and operate. First, we have seen the spectacular introduction and wide use of high–end workstations with very low prices. Second, disk units become larger and more reliable. Finally, computer network technology has matured and offers reliable operations for file transfers, remote access and message handling. The main point of this paper is to show that the continuous demand for even higher system throughput in DBMSs [ABGM90] can be achieved by combining off–the–shelf systems running on multiple but single-CPU hardware. Such generic hardware solutions coupled with the appropriate software systems are less costly and a lot more flexible. In particular, we are concerned with the performance of modern Client-Server (CS) database architectures.

All CS architectures we have studied consist of a number of workstations (clients), one or more large workstation(s) or mainframe(s) which undertake the role of the server(s), and a local area network connecting them all together. We assume that multiple databases running on different servers are autonomous and that no inter-database transactions exist. The client functionality ranges from just running the application with no caching on either main memory or disk and minimal or no decoupling between the client and the server, all the way to having full cache management capability on the client and high degree of decoupling and data distribution. Each server can have either a single or an array of disks for parallel I/O. Several CS database architectures can be built with the above functionalities. This paper concentrates on the following: (a) No caching on the clients and single disk on each server. This is the standard and minimal functionality configuration (SCS). (b) No caching on the clients but multiple disks on each server (CS–MD) for data replication and parallel I/O. (c) Enhanced disk cache management functionality on the clients for dynamic data migration and incremental maintenance of cached data, and a single disk on the server (E–CS). Downloaded and cached data provides the

[*] Also with the University of Maryland Institute for Advanced Computer Studies (UMIACS).

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

replication needed for parallel I/O. (d) As in (c) with the addition of a special purpose buffer manager dedicated for facilitating the access of the server logs for the incremental cache management (E–CS–LB). We chose for evaluation the above (b-d) configurations as they are most comparable with regard to hardware (equal number of disks and roughly equivalent replication overhead). The SCS is used as the basis for our analysis.

There are three basic classes of application databases which can be facilitated by the above architectures:

1. Mostly Read transaction (MR) databases with a lot of read–only clients and very infrequent updates. The MRT class includes typical dial–in databases such as CompuServe, libraries, software repositories, internet ftp sites, etc.
2. Constant number of Update transaction (CU) databases with many read–only users and a constant number of updates. This class includes databases such as those storing stock market quotes where only a fixed number of privileged users perform updates while the rest perform look up operations.
3. Variable number of Update transaction (VU) databases in which both reads and updates are proportional to the total number of clients. The VU class includes traditional data processing environments.

This paper is concerned with the expected performance ranges of these classes.

There is a number of studies that deal with similar issues to those we are discussing here. Hagman and Ferrari [HF86] are among the first who tried to split the functionality of a database system and off–load parts of it to dedicated back–end machines. Roussopoulos and Kang [RK86] proposed the coupling of a number of workstation-based DBMSs loosely-coupled with a mainframe DBMS. A similar cooperation between a server and a number of workstations in an engineering design environment is examined in [KDG87]. The DBMS prototype that supports a multi–level communication between workstations and server which tries to reduce redundant work at both ends is described. Rubinstein et al. in [RKC87] presented the RAD–Unify type of DBMS architecture where the server executes low level database operations (locking and page handling) while diskless workstations perform query processing and use their virtual memory to improve query response time.

DeWitt et al. [DMFV90] examine the performance of three workstation–server architectures from the Object–Oriented DBMS point of view. Wilkinson and Niemat in [WN90] propose two concurrency control algorithms for maintaining consistency of workstation cached data. Alonso et al. in [ABGM90] support the idea that caching improves performance in information retrieval systems considerably and introduce the concept of quasi–caching. Different caching algorithms— allowing various degree of cache consistency—are discussed and studied using analytical queuing models. Delis and Roussopoulos in [DR91b] examine the performance of incremental maintenance under light updates and show that this architecture offers significant increase of transaction processing. In [RD91], we give a short description of three CS DBMSs and report some preliminary results on their performance. Carey et al. in [CFLS91] examine the performance of five algorithms that maintain consistency of cached data in client–server DBMS architecture. The important distinction between this work and those mentioned above is that client cached data are maintained in main memory and are not disk resident. Wang and Rowe in a similar study [WR91] examine the performance of five concurrency control–cache consistency algorithms in a Client–Server configuration. Their simulation experiments indicate that either a two phase locking or a certification consistency algorithm offer the best performance in almost all cases. Some work indirectly related to the issues examined in this paper are the Goda distributed filing system project [SKK$^+$90] and the cache coherence algorithms described in [AB86].

In this paper, we study the trade–offs between data replication (CS–MD) and query result caching (E–CS) and the effect of updates in the CS architectures. The role of each server is to maintain consistency while multiple copies of data or cached query results facilitate parallel I/O. Our study shows that except for the case of update–only workloads the CS–MD, E–CS (E–CS–LB) architectures outperform the standard Client–Server architecture by one or more in some cases orders of magnitude. The CS–MD architecture offers performance comparable to those achieved by E–CS (E–CS–LB) for an rather limited number of participating clients. We also investigate the scale-up behavior of all these CS database architectures. To the best of our knowledge, neither analytical nor simulation study has quantified the scalability issue.

The paper is organized as follows: in section 2 we provide a detailed description of the examined Client-Server DBMS architectures. Section 3 gives the simulation closed network models for all the DBMS configurations. Section 4 discusses the results of our experiments. Finally, conclusions are found in the last section.

2 Client–Server DBMSs

The general Client–Server model for network applications [Ste90] can be easily extended to database systems. Indeed, a number of DBMS suppliers and research prototypes already follow this paradigm of computing [DBP88, KGBW90].

2.1 Standard Client–Server

In this model, a database process running on a server machine (*server process*) waits to be contacted by *client* processes. If no request is issued by the clients then the *server* process goes to sleep waiting for some request to occur. A client process opens up a communication channel and connects to the specific address of the server machine. The server process listens to the LAN and whenever it receives a message, it wakes up in order to compute the incoming request.

The processing of a request is done with the spawning of a server (lightweight) process for every client request. These newly spawned service processes as also known as *concurrent servers* [Ste90]. All the concurrent servers have to go through the system loop—interleaved CPU and disk operations. As soon as a concurrent server process completes its computation, it passes its results and/or messages through the open communication link with the appropriate client and finishes. It is the job of the client to close the communication channel and to continue the remaining of its computation while having the server information available. The same model can be applied for client and server processes resident in the same machine. In this paper, we assume that clients and concurrent servers run

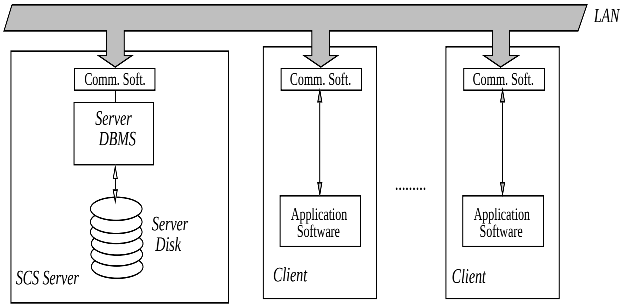

Figure 1: SCS Architecture

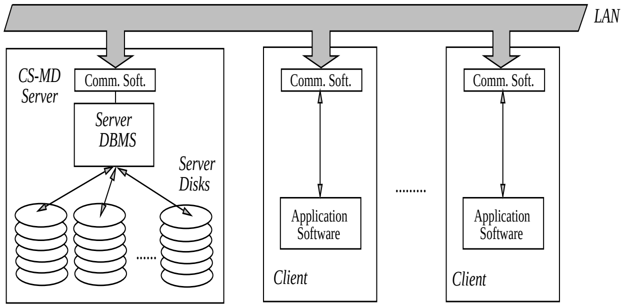

Figure 2: CS–MD Architecture

on different hardware.

The SCS architecture off–loads CPU processing from the servers to the clients. The application programs along with other interface utilities—such as the DBMS presentation manager—are run on the clients without affecting the server. The bulk of the database processing and I/O remains a server task. Figure 1 presents a cluster of clients with a single server.

2.2 Client–Server with Multiple Disks

Although parallel application program processing is advantageous, server resources and especially disk accesses constitute a great impediment into achieving high performance. The I/O congestion on the servers remains as high as in centralized single disk DBMS architectures. Thus, alternative architectures utilizing I/O parallelism must be explored.

The CS–MD architecture incorporates a large number of disks on the servers and intelligent controllers which utilize multiple heads for reading in parallel from replicated data (Figure 2). We assume that each disk has a copy of the database –full replication– for it provides a practical approach to the data allocation problem [Wol89] and does not depend on the quality of the allocation and striping methods. This configuration is very similar to RAID level one [PGK88]. The job management uses a locking mechanism similar to that of the SCS *concurrent servers* with the only exception that it uses the read-one/write-all protocol, that is, an update commits only when all disks have finished the update, and a read is done from a single disk, the one which is idle or has the lightest load. This configuration favors reads at the expense of updates but avoids the overhead of partial replication and skewed access patterns.

Apparently, there are methods for reducing the replication of data on the disk, but we are not concerned with this issue here. Disk striping and the various levels of RAID architectures proposed elsewhere [SM86, PGK88] deal with this problem but they could impose further delays for reads. We believe that the RAID level one configuration lends for direct comparison with the E–CS architectural variations. The major advantage of the CS–MD architecture is that it distributes read operations over a number of disks yielding better response times and ultimately increased system throughput. However, write operations may create additional conflicts, more blocking, and increased overhead. We will show the range in which the read benefits offset the replication write overhead. Perhaps, a more serious disadvantage for this architecture at this point is its cost, but this is outside the scope of this paper.

2.3 Enhanced Client–Server

The above two architectures centralize the database operations on the servers and distribute to the clients only application/interface processing. The Enhanced Client–Server architecture goes further and distributes to the clients a good portion of database operations. To achieve this, it utilizes the local disks available on the client workstations for caching query results once retrieved from the servers and delivered to the clients. The additional merit is that the clients' disks are accessed asynchronously contributing to greater I/O parallelism –whenever this is possible. This architecture requires a disk cache management functionality on the clients for dynamic data migration and incremental maintenance or replacement of cached data. This functionality is very similar to that of a DBMS except that a) it needs no transaction recovery and security managers (each client user runs in his/her own locally cached environment), and b) it is capable of handling cached query results which are *bound* to server(s) relations. The E–CS architecture is depicted in Figure 3.

Each database resides on a server[1]. Initially, the clients can start with either an empty local cache or with some data of their interest. Queries involving server relations are transmitted to and processed by the server(s). Their results–in the format of tuples–are shipped to the appropriate clients for displaying and/or other processing. Clients can then cache these results in local relations on their disk for later use. At that time, a binding between a client and the server is created. The binding in the format of query conditions and a timestamp is stored in the catalog of the client. Bound cached query results are the product of selections/projections/joins from the server relations. Dynamic caching permits the clients to define their "operational database space" according to their needs and constitutes a form of replication. It is reasonable to assume that most clients would be interested in a subset of the database space, therefore, the degree of replication in N clients would be less (or a lot less) than N copies (full replication) of the whole database space.

Updates are sent for execution to the server(s) which acts (act) as the *primary site(s)*. Logs are utilized for

[1] In this paper we assume that each query references relations from a single server database even though multiple servers can be accessed at any time.

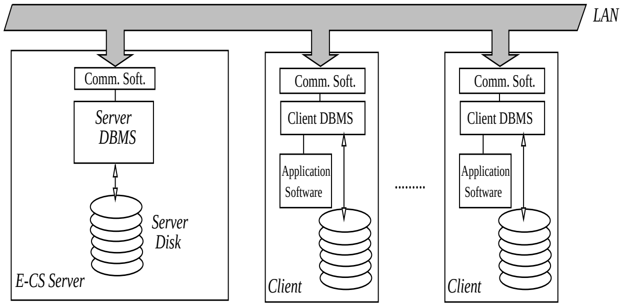

Figure 3: E–CS Architecture

the recording and the incremental propagation of updates. We assume that these logs are not the same with those used for recovery. On the one hand, this introduces additional overhead but, on the other, update propagation logs record only committed updates and are smaller and far more accessible than recovery logs of commercial DBMSs. Every server relation is associated with an update propagation log which consists of timestamped inserted tuples and timestamped qualifying conditions for deleted tuples. Each client, that caches query results, associates with every bound relation the last time (timestamp) the client received pertinent modifications from the server. These timestamps along with the client binding information are used for requesting the differentials from the update propagation logs. In this way, the incremental maintenance of the cached data is implemented. The client cache manager (catalog) is responsible for keeping track of both the timestamps of the last update it received for each cached item and the binding conditions. This releases the server DBMS(s) from such a bookkeeping which multiplies very quickly with the number of clients.

Query processing against already cached data is preceded by a request to the pertinent servers to propagate all relevant changes. The servers are required to look up the portion of the log that maintains timestamps greater than the one submitted by the client so far. From this portion, only the tuples that qualify the client binding conditions need to be transferred. Thus, only *relevant increments* of the update logs are shipped back to the clients. Deferred (lazy), periodic, or eager update propagation strategies can be employed. The set of algorithms that carry out these tasks are based on the *Incremental Access Methods* for relational operators described in [Rou91]. The transmission of differentials significantly reduces data transmission over the network as it only transmits relevant increments affecting the bound object. This is in contrast to the previous two CS architectures in which query results are continuously transmitted in their entirety.

It is important to point out some of the characteristics of the concurrency control mechanism assumed in the E–CS architecture. First, since updates are done on the server, a 2–ϕ locking protocol is assumed to be running by the server DBMS (this is also suggested by a recent study [WR91]). For the time being, and until commercial DBMSs reveal a 2–ϕ commit protocol, we assume that updates are single server jobs. Second, we assume that the update propagation logs on the servers are not locked and, therefore, the server can process multiple concurrent requests for incremental updates.

Another advantage of an E–CS type of architecture is that the client's DBMS functionality can be used for incorporating into the downloaded database portion other private relations for a value-added. Finally, from a reliability point of view, the crash of a single client workstation has no affect on the rest of the system. When, the failed client comes up, then it is brought up to date from the server's logs, in the same incremental fashion. Even when a server goes down, only the updates of that server cannot be performed. Reads can be performed from the dowloaded data and/or other servers. This is not feasible for either the SCS or CS–MD architecture.

2.4 E–CS with Buffered Logs

The distribution of database operations in the E–CS architecture depends on the incremental access of the server(s) update logs. When the number of clients becomes very large and the updates are of significant size, the logs may become hot spots. To avoid this potential bottleneck, some parts of the logs could be buffered and all the relevant look ups are directed through the buffer first. The parts of the log that can not be found in the log buffer area are retrieved from the server's disk. The potential benefit of this approach is that with a very modest amount of buffer space we can improve the performance of the E–CS architecture substantially. This configuration, which is a slight extension of E-CS, is termed Enhanced Client Server with Log Buffers architecture (E–CS–LB). In the worst case scenario, the E–CS–LB should demonstrate approximately the same advantages with those of E–CS. This would occur when updates are very large or too many for the buffer to retain the working set.

3 Client–Server Models

In this section we describe the models used for the simulations. We build them in an stepwise manner by discussing first database operational parameters, network, buffer and log model features. Finally, we outline the developed queueing models for measuring performance of the various CS architectures.

3.1 DBMS Operational Aspects

In order to achieve a fair comparison among the different variations of the Client–Server architectures, we use a fairly standard set of parameters for the SCS architecture (see Table 1), and a slightly extended set to account for the enhanced architectures. Disk access times follow a uniform distribution with averages shown in Table 1. Every time a read from or write to the disk occurs a number of instructions are executed by the appropriate CPU (*read_a_page*, *write_a_page*). We also distinguish the CPU required by DBMS for performing a selection, a projection, a join, and an update on a page that is already resident in main memory. Issues of DBMS buffer management on the servers [CD85] are not considered in our models, although they could be incorporated with a reasonable effort.

Database requests are submitted by the clients in the form of jobs. Each job consists of a DBMS operation (such as select, join etc.) on either base relations or cached query results. All these requests are enclosed in a pair of BEGIN_JOB and END_JOB control statements. Every job can be either of *read* (select, project, join) or *write* type (insert, delete, or update). In [WR91], it is shown the 2–ϕ

(phase) locking protocol for server transactions gives system performance comparable to those attained with more elaborate ones. Thus, we use this standard 2–ϕ locking protocol for maintaining consistency on the servers.

Deadlock detection is done by maintaining a wait–for–graph of all active processes during their execution. A time–out mechanism triggers a deadlock detection algorithm in which cycles in the graph of deadlocked processes are discovered and one or more processes in a cycle are preempted in order to allow the other(s) to continue their processing. Each time the deadlock detection algorithm is run, a dd_search msec per active process is charged, and if a deadlock is found a standard $kill\_time$ penalty is charged to the total CPU processing for finding and killing the least advanced job. The killed process is restarted after a delay which is proportional to the number of active jobs in the ready queue of the server. In order to avoid a potentially large number of restarted jobs, we restrict the maximum number of jobs that can be active at the same time on the server (multiprogramming level or mpl).

$DBMS\ Operational\ Aspects$	$Value$
$page\_size$	2KBytes
$serv\_disk\_acc\_tm$	12 msec
$cl\_disk\_acc\_tm$	16 msec
srv_cpu_mips	28 MIPS
cl_cpu_mips	20 MIPS
$read\_page$	3500 ins
$write\_page$	7000 ins
$inst\_sel$	10000 ins
$inst\_prj$	11000 ins
$inst\_join$	29000 ins
$inst\_mod$	12500 ins
$inst\_log$	9000 ins
mpl	12
bl_delay	1msec
dd_search	10 msec
$kill\_time$	200 msec
$Network\ Features$	$Value$
rpc_{del}	10 msec
net_rate	10 Mbits/sec
$Caching - Logging\ Characteristics$	$Value$
α_{Rel_i}	0.5
$Write\_Log\_Fract$	0.10
$buffer\_size$	0.4 MB

Table 1: Various model parameters and their values

3.2 Network Model

Our network model is a rather simple means used predominantly for two purposes: a) to route messages and acknowledgments between the server and the clients/workstations, b) to transfer to the corresponding client sites query results in the case of SCS and CS–MD, and/or incremental updates in the case of E–CS and E–CS–LB. The cost associated with the use of the network has been measured in a recent experimental study [PJ91] approximately 3 msec for exchanging very short control messages between two workstations under relatively uncongested network. We use a higher network overhead to account for the more substantive messages and context switching cost rpc_{del} per message [Hal89]. In addition to this overhead, we charge data movement at a transfer rate of net_rate (Table 1).

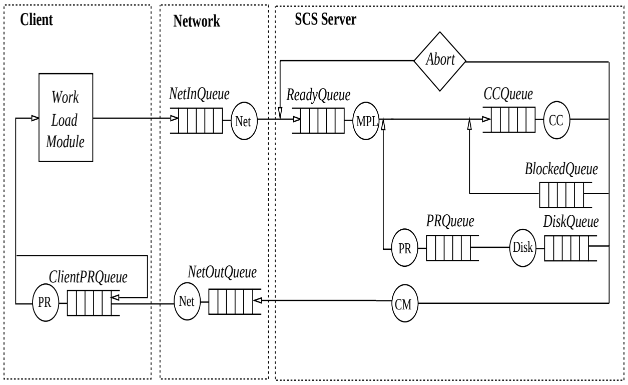

Figure 4: SCS Model

3.3 Client Data Caching, Logging, and Log Buffer Management

The E–CS clients usually start with no data cached from the base relations. As time passes, they incrementally cache on their disk query results that are of interest to them. In our simulations, we assume that during an initial phase a portion of the base relations data is cached on the client disks. The portion of every base relation that is cached at this initial phase is represented by α_{Rel_i} where i is an index of a relation. Obviously, this factor ranges from zero to one and it can be specified individually for each client (Table 1).

E–CS updates are charged not only with page writes into the base relations but also with additional page writes into the update propagation logs. For each committed job, the log write charge is a percentage of the corresponding page writes into the base relations (assuming that a dirty page does not have all its tuples changed). This percentage can be specified individually for each client ($Write\_Log\_Fract$). Logs are maintained in a strictly serial and append only fashion. For every log page $inst\_log$ instructions are required for this page to be processed.

In the case of E–CS–LB, a buffer area in main memory is maintained in the server to hold exclusively log pages. This area is totally different from the buffer areas used by the server DBMS. At the very beginning all this space is free and gets loaded as log pages are read for the propagation of increments. The replacement of the pages is done in a FIFO discipline and there is no discrimination against or for some specific relations of the database (global replacement strategy). The size of this buffer is $buffer\_size$ set to 200 pages for the first part of the experiments. This buffer management favors relatively up to date cached query results whose increments are still in the buffer area. Very outdated results whose updates fall outside the buffer window are charged with the necessary I/O to read the log pages, as in the E–CS architecture.

3.4 Simulation Queueing Models

In Figure 4, a model for the Standard Client–Server architecture is shown. It consists of three parts: a) the client nodes (only one is depicted for brevity) b) the network manager and c) the server. The client nodes are made up of two components, its processing unit ($ClientPRQueue$, PR) which enables the client to run

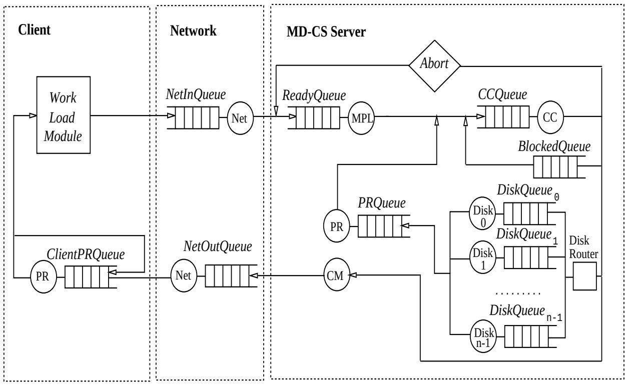

Figure 5: CS–MD Model

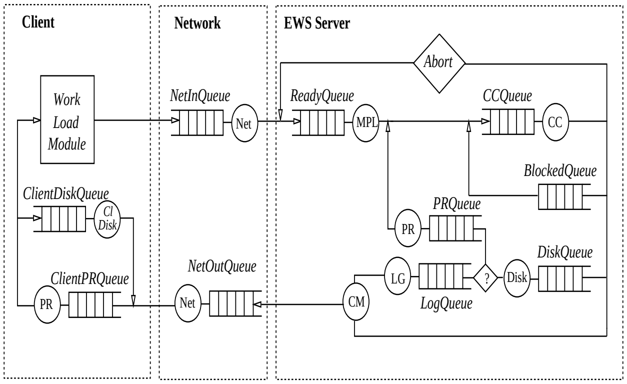

Figure 6: E–CS Model

application programs, and the work–load module that is the element responsible for the generation of the submitted jobs. The format of the model interface allows for different work–loads to be formulated. The network manager model consists of two queues that forward messages from the clients to the server ($NetInQueue$) and transfer results and messages from the server to the clients ($NetOutQueue$) [FD90].

The structure of the server model presented in Figure 4 has been influenced by previous similar models such as those used in [ACL87, WN90]. Jobs enter the $ReadyQueue$ and wait at this point until the server admits them for processing. A maximum number of concurrent jobs is allowed to avoid large number of job restarts and control data and resource contention. The MPL processing module oversees the number of the active jobs. The additional jobs wait in the $ReadyQueue$ until they become active in which case they advance to the $CCQueue$ (Concurrency Control Queue).

The role of the CC processing module is multiple. First, it handles job requests for disk pages. In order to retrieve them the appropriate locks need to be acquired. Locks are of two types: exclusive or shared. The standard lock compatibility matrix is being used [GLPT76]. We assume that the lock table is main memory resident, and therefore, there is virtually no overhead for the processing of the locks. Once the lock requests have been processed, there are several paths a job may take. The first is that all the required locks are obtained and the job is queued at the $DiskQueue$ for service by the data manager. The second is that the job is blocked due to a lock conflict and has to wait until this conflict ceases to exists. Thus, the job enters the $BlockedQueue$ and after spending some time in there (*bl_delay*) re-enters the $CCQueue$ for another lock request trial. Every time a blocked job is redirected to the $BlockedQueue$ a time out counter is incremented. As soon as the value of this counter reaches an upper bound, a deadlock detection algorithm is invoked by the CC processing module. If a cycle is found in the wait–for graph a job to be killed is selected and aborted. This decreases *mpl* by one and the killed job joins again the $ReadyQueue$ of the server. If a job is about to commit (finished all the reading, processing and writing back to the disk) then it is directed into the CM module that releases all the job locks and queues the result into the $NetOutQueue$ for transfer into the proper client.

The data manager consists of the $DiskQueue$ and the $Disk$ processing unit. The $Disk$ module "charges" time for the disk operations performed by the server. As soon as job pages have been retrieved from the disk, they are placed into the processing queue ($PRQueue$). The PR processing module "charges" the CPU with the appropriate amounts of time for the different types of DBMS processing (selection, projections, etc). After that point, jobs enter again the $CCQueue$ to continue their lock and page request cycle until they finish.

In Figure 5, the Client–Server with Multiple Disks model is presented. The generation of the jobs and their admittance into the server is done in exactly the same way described for the SCS model. The only difference is in the way the data manager has been constructed in this case. A number of n disks constitutes the server data manager. The disk router is the element that decides on what $DiskQueue_i$ a current retrieval request needs to directed to. Although read jobs go through one disk only, write jobs require to flush their results in all disks. Writes commit as soon as all the disk units have enforced their updates for a particular update job.

Figure 6 shows the closed network model for the Enhanced Client Server architecture. There are two fundamental differences from the model of Figure 4. The first is that all the workstations (clients) have been provided with a local disk to facilitate cached data and have a DBMS running locally which occasionally interacts with the server database. Therefore, the client model has been changed slightly to include a disk unit made up of a queue ($ClientDiskQueue$) and a local data manager $ClDisk$. Once all required increments are received from the server and written into the local disk, the processing of the query (initiated by the work–load module) may commence. All the processing time (CPU and disk time) is charged on the local resources.

The second difference is that the structure of the E–CS server has been extended in order to reflect the incremental DBMS accessing. Every time a client sends a request, certain sections of the logs need to be retrieved and processed in order to decide if they are relevant to the work done by that client. Write type of jobs follow the processing route at the server model as explained in the SCS model. After a read job enters the server "core", it is the task of the

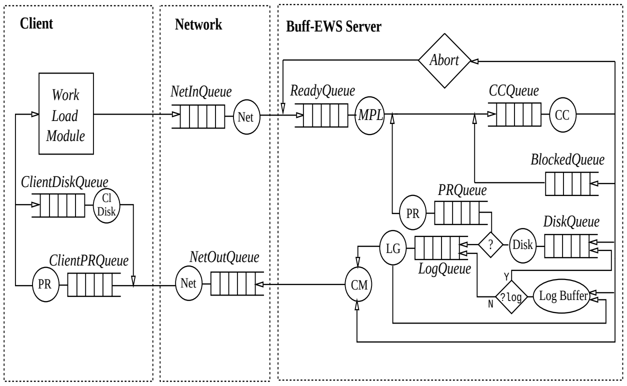

Figure 7: E–CS–LB Model

CC processing unit to direct this job to the disk queue for log page retrievals. Thus, there is a differentiation in the type of pages the data manager retrieves: they are either pages required for the processing of updates or log pages. The former are directed through the *PRQueue* and finally via the processing *PR* module back to the *CCQueue*. The latter through the *LogQueue* are processed by the *LG* processing unit and the filtered portion of the retrieved log (increment) is directed via the *NetOueue* to the proper workstation. The *CC* module in addition to the usual processing has to perform an extra operation at the commit time of a job, that is to write the updates–in the format of insertions and deletions–in the appropriate log area.

Figure 7 shows the enhanced E–CS model with log buffering capability for the server (E–CS–LB). As soon as the concurrency control mechanism determines that some increment is required, the buffer area is looked up in order to determine if a portion of the log is already there. If found then it is directed to the *LogQueue* for the proper processing. Otherwise, the log is accessed from the disk. Log pages are processed by the *LG* module (Table 1) and those retrieved from the disks are placed into the buffer area using FIFO replacement policy if required.

The simulation packages for all four models were written in C and their sizes vary from 4.8K to 5.4K lines of source code.

4 Simulation Results

This section presents our evaluation methodology and major simulation results.

4.1 Measurement Methodology

Workload modeling is one of the most important aspects in any evaluation study [Fer84]. During this modeling the major system parameters are identified. We range the values of these parameters in order to reveal the operational advantages and disadvantages of the systems under examination during the measurement phase. In our models, there are four groups of such important parameters. Namely, those concerning with: 1) the hardware (i.e. CPU power, disk access time, number of disks etc.), 2) the logical processing system (i.e. multiprogramming degree, DBMS operation costs, etc.), 3) the data resources (i.e. number and size of server relations, data sharing, replication, percentage of workstation cached server relations etc.) and 4) the client dynamic patterns of data access that deal with the system job mix. In our experiments, we have tried to address all of the above groups of parameters but the main emphasis was given to the last item. The means to specify the client data patterns of access is that of *job streams*. A job is either a query or an update. A *job stream* is a sequence of jobs made up by mixing queries and updates in a predefined proportion. In the two extreme cases, we can have either query or update only streams. Every client is assigned to execute such a stream.

Using the stream to vary the query/update ratios and according to the database classification given in the first section, we run two families of experiments: 1) those with Constant number of Update jobs (CU), where a constant number of four streams submit updates and the remaining clients queries only. 2) Those with Variable Update jobs (VU) where each stream is a combination of both queries and updates—updates constitute 10% of all the jobs and are uniformly distributed over the queries. We also experimented with two types of queries: Large Read jobs (LR) that have read page selectivity of up to 30% of the base relations and Small Read jobs (SR) with page selectivity up to 10%. Updates are carried out on the server base relations with page selectivity varying from 0% to 8%. The update page selectivity specifies the percentage of the base relation pages which have to be written by an update. The 0% update selectivity streams corresponds to MRT databases.

Four experiments were conducted using job streams corresponding to: CU-LR, CU-SR, VU-LR and VU-SR. In the simulations, we vary two parameters : the number of participating clients from 10 to 250 and the update page selectivity from 0% to 8%. The simulators create streams by randomly selecting jobs from sets of query and update templates. The page update selectivity remains the same throughout all the modifications of the same job stream. The number of participating jobs per stream was selected to be long enough (80) so that the systems reach a stable state before finishing with confidence of more than 96%. The same exactly streams were submitted in all CS configurations.

The main performance criterion for our evaluation is the overall average job throughput of the various CS configurations. The average throughput is defined as:

$$T_i = \frac{Number\ of\ Jobs\ Completed}{Average\ Completion\ Time\ for\ all\ Clients},$$

($i = SCS, CS\text{-}MD, E\text{-}CS, E\text{-}CS\text{-}LB$) where the completion time for a client is the commit time of the last job of its stream. The average throughput is measured in jobs per minute (JPM). We also use log buffer hit ratio and network utilization to analyze some of our results.

In the first set of experiments, client think time (which is the time between the completion of one job and the submission of the next) is set to 0. The rational for this is that we wanted to test the CS architectures under stringent conditions. In a later subsection, we present some additional results as we increase the think time. We also examine the performance of the configurations under pure update loads, representative results from the network utilization and the performance of E–CS–LB under increased buffer sizes.

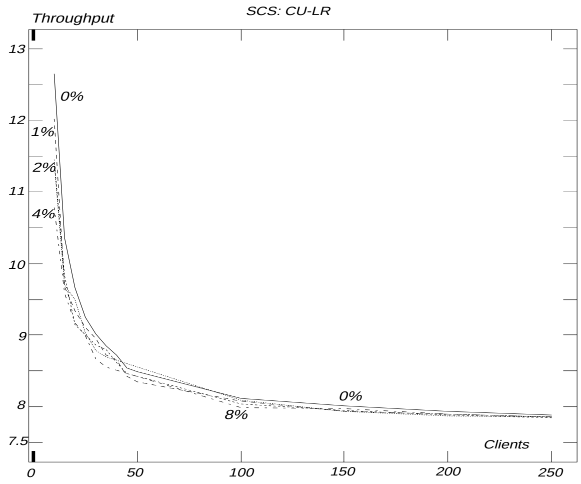

Figure 8: CU–LR Experiment in SCS

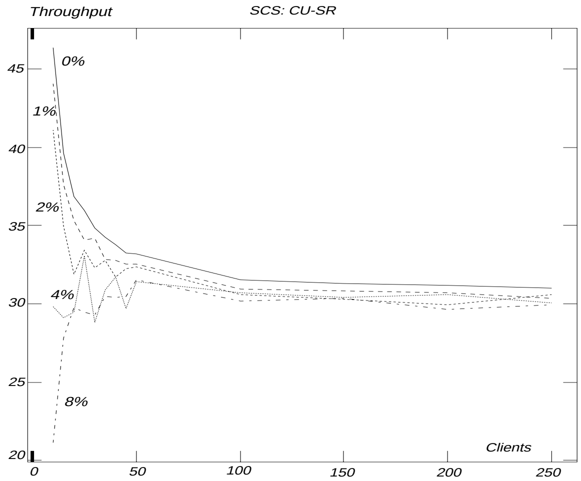

Figure 9: CU–SR Experiment in SCS

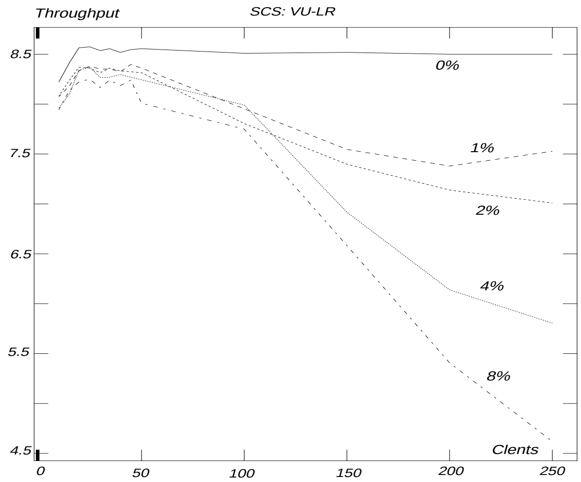

Figure 10: VU–LR Experiment in SCS

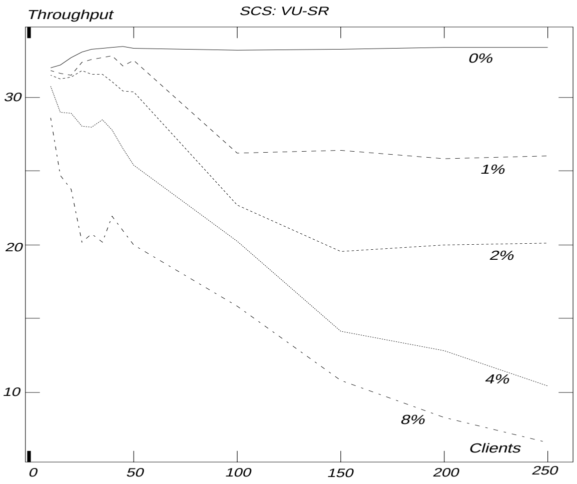

Figure 11: VU–SR Experiment in SCS

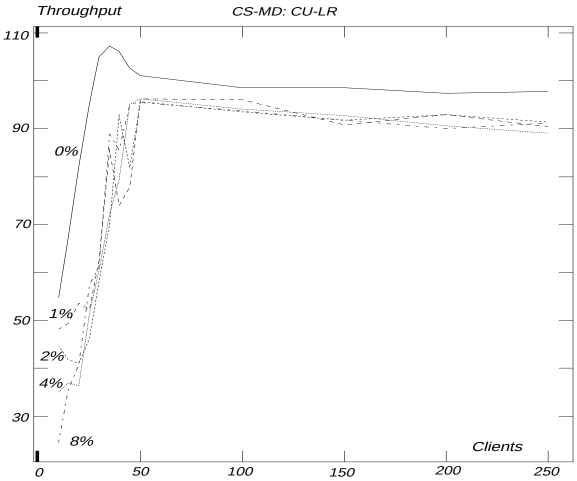

Figure 12: CU–LR Experiment in CS–MD

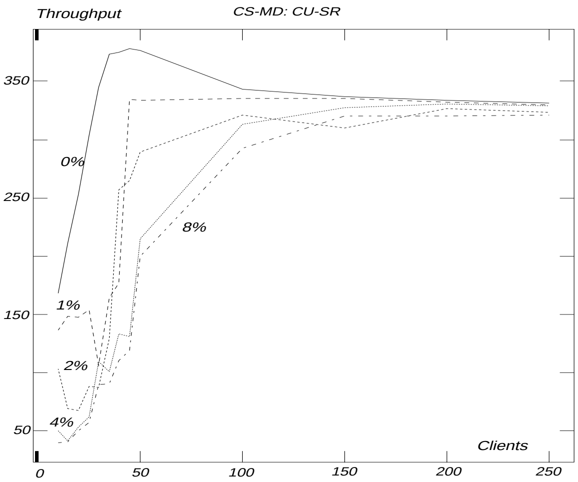

Figure 13: CU–SR Experiment in CS–MD

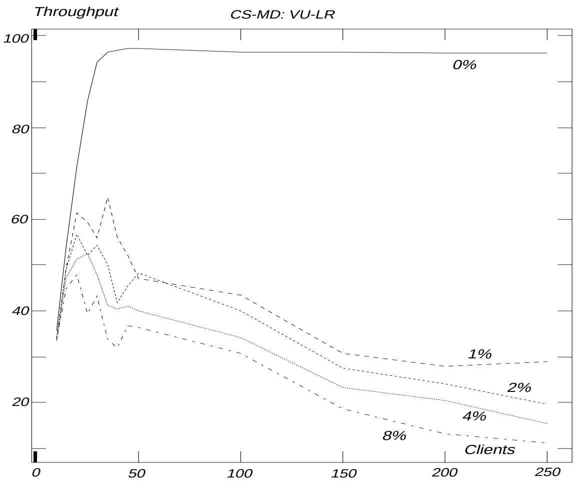

Figure 14: VU–LR Experiment in CS–MD

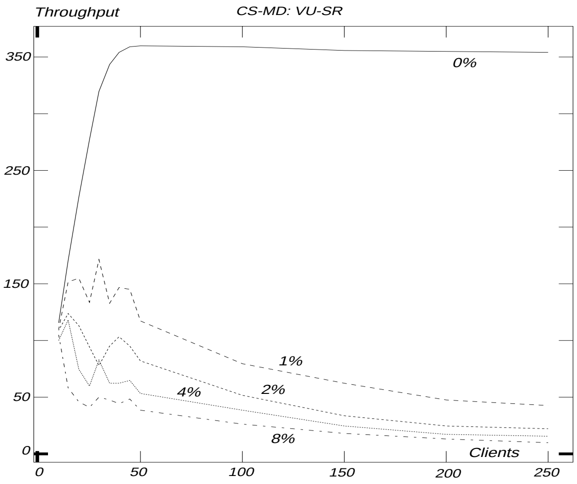

Figure 15: VU–SR Experiment in CS–MD

4.2 Experiments and Analysis

All simulations measure the performance of one server with a variable number of clients. Such a cluster provides the basis for comparison. Figure 8 shows the average throughput rates for the CU–LR experiment on the SCS architecture. Five curves are graphed each one corresponding to streams with update selectivities 0%, 1%, 2%, 4% and 8%. The highest throughput is obtained when a relatively small number of clients is involved and varies from 12.7 JPM for the 0% curve to 8.6 JPM for the 8% curve. Overall, the performance of the non–zero curves follows closely that of the zero curve throughout the range of the experiment. We observe a steep decline in the performances for the area between 10 and 35 clients. This occurs because the fraction of the number of relatively short lived updates over the number of queries in this area is significant. For more than 50 clients, we observe that all the curves stabilize in the same approximately throughput rate of 8 JPM (high disk utilization area). Figure 9 shows the results obtained by using small queries but the same updates as in Figure 8 in the client streams (CU–SR). The achieved throughput rates are higher than those of Figure 8 since the size of the submitted queries is significantly smaller. Again beyond 50 clients, rates stabilize around 30 JPM.

Figures 10 and 11 show the throughput rates for the VU–LR and VU–SR experiments on the SCS configuration. For up to 100 clients, we see no significant degradation of JPM in the LR case contrary to Figure 8. After that point, the performance deteriorates faster and faster as the update page selectivity increases. The same phenomenon is observed in the SR case (Figure 11) from almost 50 clients. Both blocking and restarted transactions are responsible for the rapid configuration performance deterioration. This confirms the analytical results reported in [TR91]. Overall in the VU experiments with up to 100 SCS clients, we achieve somewhere between 7.5 and 8.5 JPM for the LR case and between 16 and 34 JPM for the SR case.

Figure 12 presents the results of the CU–LR experiment on the CS–MD configuration. The 0% update selectivity curve provides the best performance as expected. For more than 30 clients, it shows a small decrease and later on a stabilization around 98 JPM. This is somewhat surprising since one would expect for a linear increase in the performance given the non-blocking nature of these transactions (effectively no database modifications) and the increasing number of disk units. There are three major reasons for not showing this type of behavior. The most serious delays are imposed by the server CPU that reaches saturation state for more than 100 clients. Second, the size of the query answers is large and increase the network utilization drastically for more than 100 clients(ranges from .77 to .81). Finally, the multiprogramming degree set at 12 imposes a limit on the number of processes working concurrently at the server site (in the next section we present some experiments on this factor).

For the non–zero update rate curves there are clearly three regions identified: 10–25, 30-50, and beyond 50 clients. In the first region, the curves are distinct and we can see that the larger the updates are the more penalties are imposed on the throughput (the role of the writes is dominant in this region). In the second region, there is no clear winner and the curves are typically "mixed" as a result of deadlocks. In the region beyond 50 clients, the throughput rates tend to converge to around 90 JPM. We observe no more "anomalies" because the queries dominate the updates and the effect of the latter is being diluted in the cost of the former. Figure 13 shows the results of the CU–SR experiment.

Figures 14 and 15 present the results for the VU–LR and VU-SR experiments respectively. There are two points to be made: the first is that the 0% curves reaches a high point at about 50 clients and then remain approximately at the same level with values similar to those of the 0% curves of Figures 12, 13. The 8% update curve deteriorates almost as rapidly as the SCS and at 250 clients drops to only two times higher throughput than the corresponding SCS. This is the result of the nature of the disk operations (read one–write all) and the linearly increased number of updates with the number of the clients. In the VU–SR experiment and for the region of more than 150 clients, the performance values of the 2%, 4%, and 8% update selectivity curves are either about the same as their SCS counterparts or sometimes worse (at 250 clients).

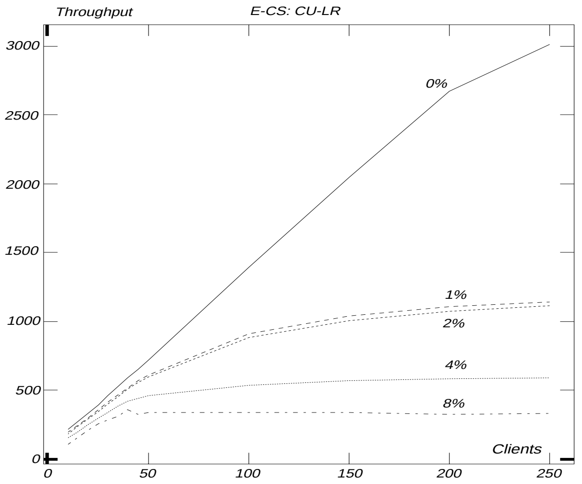

Figure 16: CU–LR Experiment in E–CS

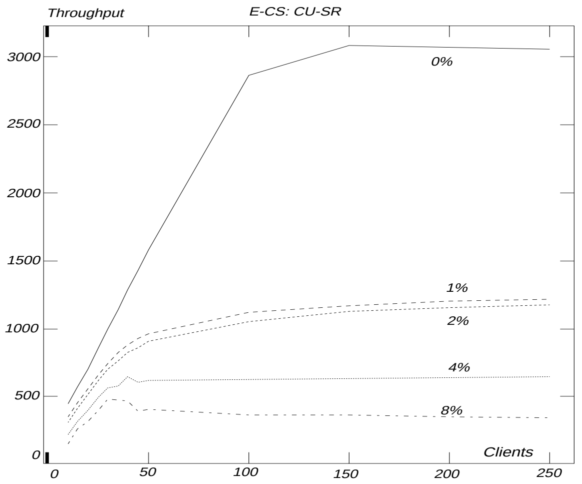

Figure 17: CU–SR Experiment in E–CS

Overall for the CS–MD configuration, we observe roughly a 10–fold improvement over the SCS for the CU experiments. Because the number of updates is constant the use of multiple disks offers significant improvement. However, in the VU experiments, the linear increase of the updates diminishes much of the benefit to very moderate improvements of less than 4 times at about 100 clients and almost no improvement in the area of 200 clients and above. The increased blocking caused by the 2–ϕ locking dictates the maximum throughput and clearly demonstrates that no further improvement can be expected under this concurrency protocol.

Figure 16 shows the E–CS performance results for the CU–LR experiment. The 0% update curve indicates that the throughput of the system increases almost linearly with the number of the clients attached to the server. This is due to two reasons: 1) the clients use their local disk units to answer queries based on the already cached data (the portion that has been cached in the initial phase of the experiment which is 50% of the server base relations) and 2) the server carries out negligible data operations (no updates processed at the server site), no log maintenance operations, and handles only request and acknowledgment messages routed through the network. It is worth noting that the 0% curve presents some decline for more than 200 clients. This is a clear indication that the network gets saturated. Indeed, the network utilization increases from about 0.191 in the case of 50 clients to 0.758 for 200 clients and 0.880 for 250 clients. As the update rate of curves increases, the throughput increases for the 1%, 2% and 4% curves for up to 100 clients. After that point, we see that the performances for all obtained curves are leveled off and they remain at approximately the same levels throughout the experiment(250 clients). This happens because no server or client resource has reached its maximum utilization for the whole range of the experiment. For 100 clients, the E-CS achieves 334 JPM for the 8% and 1392 JPM for the 0% curve. Comparing with MD–CS, this represents a many–fold improvement that ranges from 3 to 14 times respectively.

Figure 17 depicts the results for the CU–SR experiment. There are two major differences with the curves of Figure 16. The 0% update rate curve "breaks" its almost linear growth at the point of 100 clients a lot earlier than before. This is due to the shorter turnaround time of the small queries and the lack of think time. Although shape–wise the non-zero update curves remain similar, the actual throughput rates are higher (at 100 clients 2866 JPM for the 0%, 1054 JPM for the 2% and 367 JPM for the 8% curve). Note, that throughput rates for all curves increase with steeper slopes than those of Figure 16.

Figure 18 shows the results for the VU–LR experiment on the E–CS architecture. The 0% update rate curves show almost the same behavior with its counterpart of Figure 16. The non-zero curves (except that of 8%) indicate the achieved throughput rates increase for up to approximately 50 clients. After that point throughput rates decrease and they seem to asymptotically follow the horizontal axis of the graph. The decline happens because of both the high server disk utilization (reaching almost 100% for all curves and for more than 50-70 clients), and the heavy use of the server CPU (utilization averaging at 82%). The maximum performance points for the non–zero update rate curves are termed "maximum throughput threshold" (*mtt*) points [DR91a] and specify the end of the regions in which we obtain almost linear performance with the number of the clients. These *mtt* points are dependent on the update selectivities as well as the query/update ratio. They fall in the area of 35 to 50 clients for our experiments. Apparently, the increased number of updates in the VU–LR case limits significantly the performance of the system in comparison with that of the CU–LR experiment. Even so, the E–CS achieves 236 JPM for the 1% curve which is 12 times better than the corresponding rate of the CS–MD configuration for 250 clients. The 8% curve at 250 clients maintains 58 JPM or 5 times better performance than its corresponding MD-CS value. The E–CS gives this improved performance for predominantly two reasons 1) the off–loading of the server disk operations and 2) the parallel access of the client disks.

Figures 20, 21, 22, and 23 depict the results of the four experiments in a E–CS–LB configuration with a server log buffer area of 0.4 MB (200 pages). The benefits for all the non–zero update curves are apparent in all graphs. In the CU experiments, the 1%, 2% and 4% curves almost touch

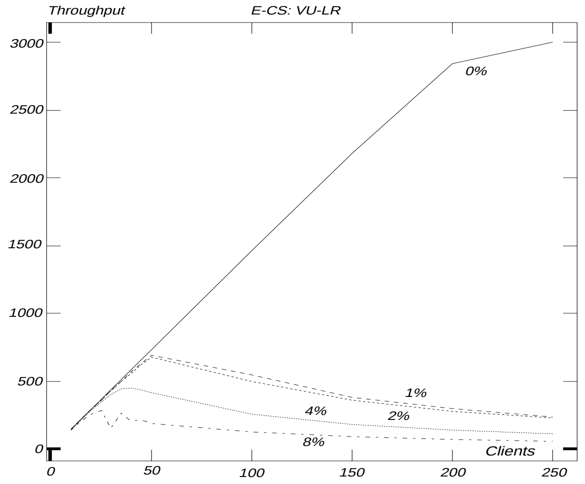

Figure 18: VU–LR Experiment in E–CS

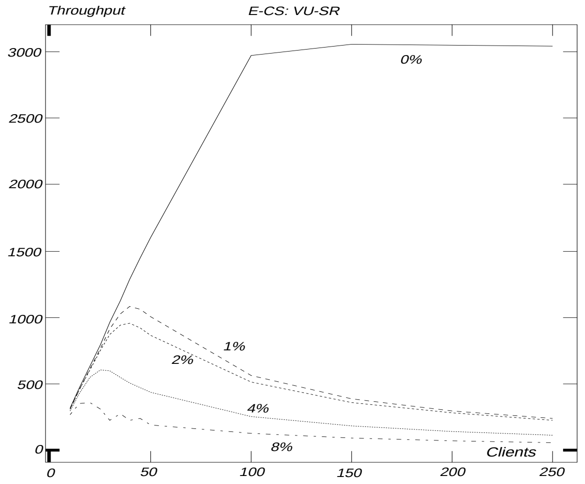

Figure 19: VU–SR Experiment in E–CS

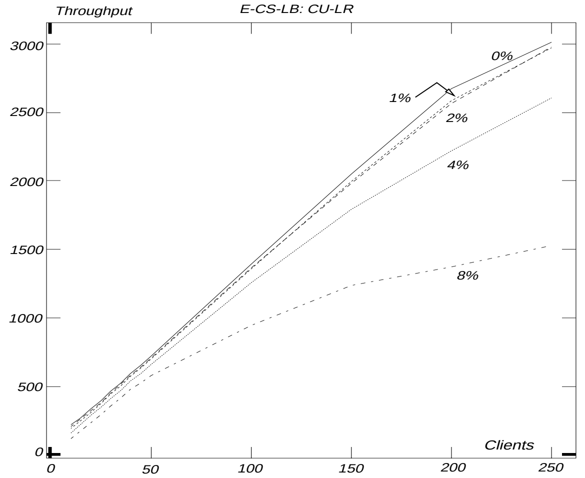

Figure 20: CU–LR Experiment in E–CS–LB

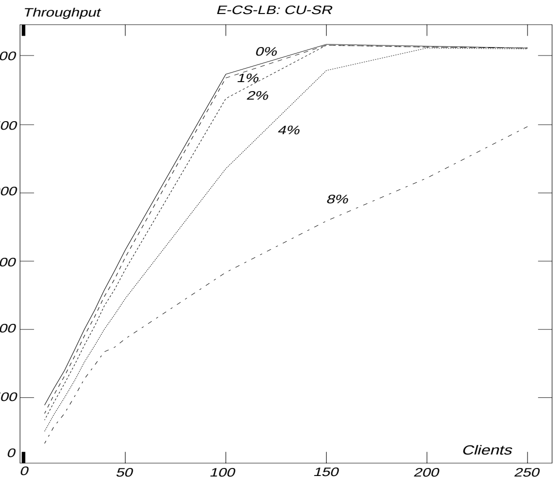

Figure 21: CU–SR Experiment in E–CS–LB

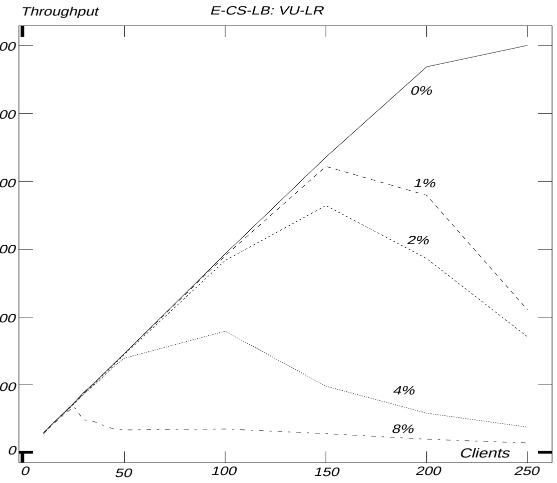

Figure 22: VU–LR Experiment in E–CS–LB

the 0% curve while the 8% curve moves up considerably from its previous positions. Note that in Figure 21, the 1%, 2%, and 4% for more than 150 clients behave similarly to the 0% curve due to the network bottleneck. The gains are more remarkable in the VU experiments (Figures 22 and 23). The simulation showed that the major reason for the E–CS performance declination was the heavy access of the disk based log. Once a buffer area for the logs is provided, a lot of disk accesses are avoided even with a very modest size log buffer. In these experiments, we observe very small improvement for the 8% curve. This is due to the fact that the size of the modifications are substantial and the buffer size can not accommodate it. Similar behavior is observed in the low update rate curves for the region beyond 200 clients. Although the modifications are relatively small, the increased number of clients creates a significant number of page faults as Figure 24 indicates.

Figure 25 shows the utilization of the network for the three configurations (SCS, CS–MD, E–CS) and for two update page selectivities: 0% and 8% (CU–SR). In the SCS case, we see some small increase in the system network uti-

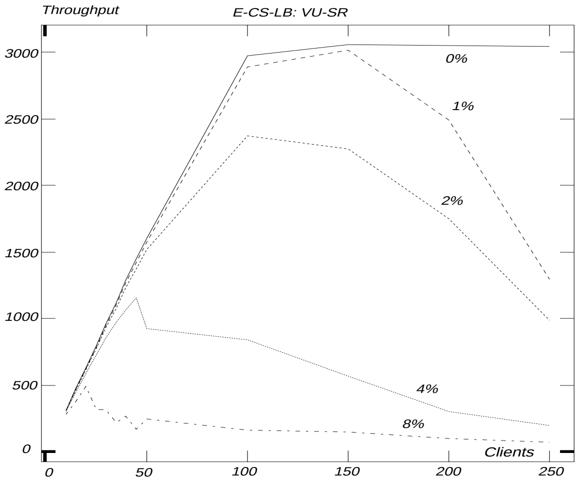

Figure 23: VU–SR Experiment in E–CS–LB

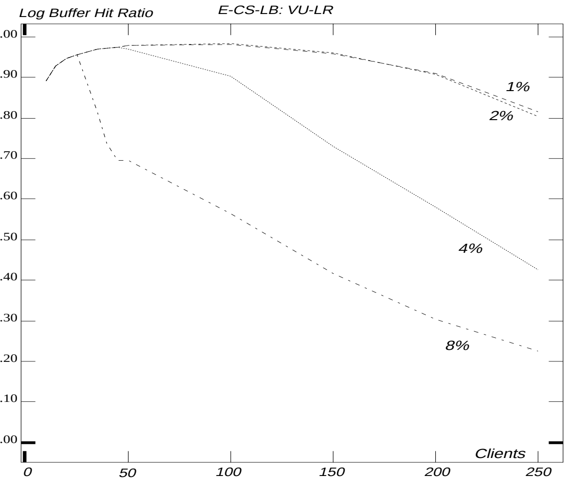

Figure 24: E–CS–LB Buffer Hit Ratios for VU–LR

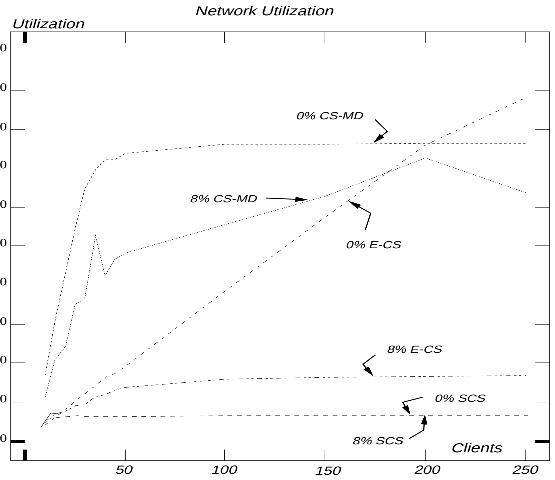

Figure 25: Network Utilization Chart

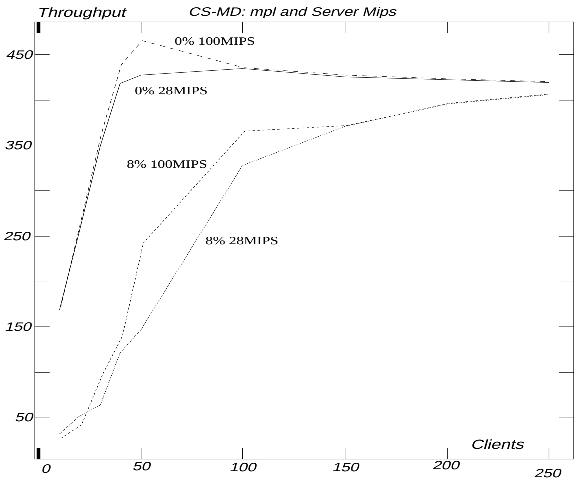

Figure 26: MD-CS with mpl=48 and CPUmips=28,100

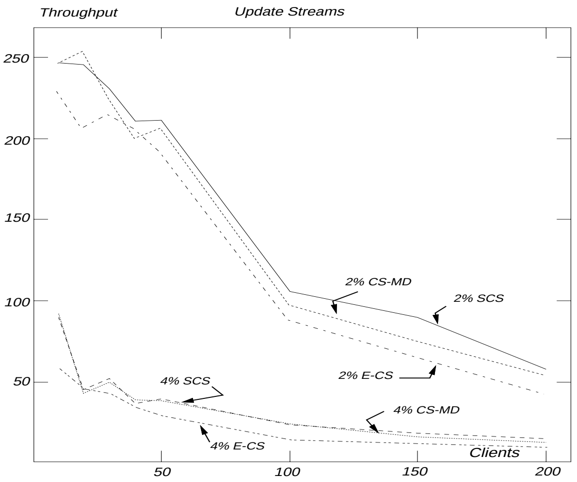

Figure 27: Update Only Streams

lization in the area between 10 and 20 clients and then the utilization remains constant. For more than 20 clients, utilization remains at about 6% because the server disk bottleneck does not allow faster production of query results. The 0% curve of the CS–MD configuration increases for up to 50 clients and then it remains stable. This is the point where the CPU becomes highly utilized (around 97%) and the network is another bottleneck (almost 80% of utilization). The parallelism achieved in the disk retrieval operations contributes to shorter turnaround transaction times and results are queued in the network queue in a much faster way than that of the SCS configuration. The blocking and the number of aborted transactions give a much smaller network utilization between 30 and 200 clients for the 8% update curve. The E–CS 0% update rate curve offers a network utilization increased in an almost linear manner with the number of clients. For the 8% E–CS update curve, we can see that the use of increments has significantly lowered the utilization of the network if compared with the corresponding curve of the CS–MD configuration.

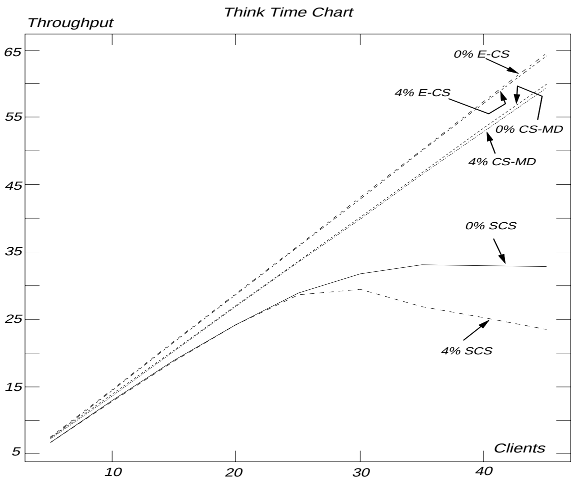

Figure 28: Use of Think Time

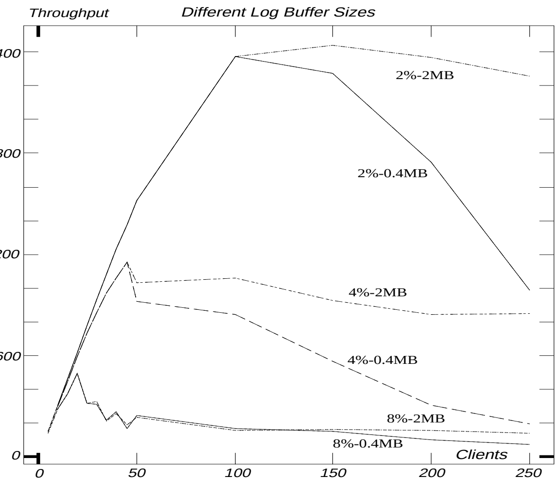

Figure 29: Log Buffer Area with 2 MB

4.3 Other Cases

In this section we briefly describe some experiments that explore the behavior of the configurations under artificially skewed circumstances.

Figure 26 presents the results of the CU-SR experiment run on a CS–MD configuration with the multiprogramming degree increased from the initial value of 12 to 48. This gives the opportunity to more processes to compete for the available system resources (especially the disk units which contribute to the longest transaction delays). In the graph of the Figure 26 there are two curves that correspond to this experiment namely those tagged “0% 28MIPS” and “8% 28MIPS”. The first curve shows clear improvement over its counterpart of the Figure 13. The second curve shows also some minor improvement as well and it finally stabilizes around the area of 405 JPM. We run the same experiment using multiprogramming 100 but we did not observe any serious improvement in the CU-SR. On the contrary, the non-zero update curves in the VU type of experiments experienced certain setbacks due to the large number of deadlocks that occurred. We next increased the server CPU power to 100 MIPS in order to alleviate one of the major bottlenecks and obtained the curves labeled “0% 100MIPS” and “8% 100MIPS”. As one can observe, the improvement is insignificant. Thus, we ascertain that the central control of disks units with an extremely powerful CPU suffers from a scalability problem.

We then turn our attention to pure update workloads. Streams were made up of update batches. Figure 27 shows the results of this experiment for all configurations and two update selectivities: 2% and 4%. The curves are falling very close together for the two different update rates with SCS having the slightly better performance. E–CS does not only need to commit the updates but also to write the appropriate log pages. This clearly indicates that all configurations are comparable in processing update–only transactions.

Think time has been considered zero in all the experiments so far. Figure 28 shows the results of the CU–SR experiment with the exception that the think time between transactions is distributed uniformly with an average of 40 secs. Only two curves are graphed per configuration (0% and 4% update rates). The experiments are shown for up to 45 clients since beyond that point the behavior of all the curves is similar to those observed in the previous section. It is worth noting that between one and twenty clients the models give very comparable results. However, beyond twenty clients the CS–MD and E–CS are the clear winners.

The last experiment we present measures the effect of log buffer size in the E–CS–LB configuration. We compared the results of the CU–SR experiment using 1000 buffer pages (2 MB) with those obtained with 200 pages (Figure 29). Only three curves are illustrated (2%, 4% and 8%) and have been superimposed on the corresponding curves of Figure 23. Throughput rates of the E–CS–LB system improve in certain high client regions. CPU utilization on the server increases as well (since more pages were present in the log buffer area). The biggest gain is observed in the 2% update rate curve whose throughput at 250 clients from about 1000 JPM goes up to almost 2250 JPM. Similarly impressive gain is observed for the 4% update curve. However, the gain for 8% update curve is insignificant. This points out for such large update page selectivity a much larger buffer is necessary.

5 Conclusions

In this paper we presented and evaluated three Client–Server architectures under multiple job streams of different complexity and varying update rates. Essentially, these architectures extend the memory hierarchy and capitalize on the availability of client CPUs to off–load data processing from the servers. Although it is possible to store very large volumes of data in each server, this approach does not scale up well. The narrow I/O bandwidth remains the stumbling block. The multiple level memory hierarchies examined here alleviate the I/O problem of a single site and create a greater bandwidth for data handling. The CS-MD achieves shorter disk seek times using multiple disks with replicas of data. The E-CS type of architecture offers an effective solution to the scalability problem. Its extended multiple level memory hierarchy (client main memory–client disk space–server main memory–log buffers–server disk space) can accommodate a larger number of clients.

Our study indicates that the standard Client–Server architecture has inferior performance in almost all cases unless mostly updates are submitted by clients. In all other cases, CS-MD and E-CS outperform SCS, some times by orders of magnitude. In the lower range of clients (10–100), the performance of CS–MD and E–CS are roughly comparable with a slight edge for the E–CS. Beyond that area though, the E–CS offers much better performance. This is mostly due to the increased I/O bandwidth attained by the parallel access of the cached data. The simulations revealed that under the presence of many clients the log buffer of the server further improves the performance of the E–CS configuration.

Further performance enhancements for the Client–Server architectures can be obtained by having multiple servers with load balancing of client requests. However, the ultimate limit is the serialization dictated by the locking protocol. Beyond such a point, performance and scalability could only be increased by more liberal concurrency protocols than the strictly serializable ones.

Acknowledgements: We would like thank Christos Faloutsos and Timos Sellis for many useful discussions and their suggestions as well as Helen Papadopoulos and Tripti Sinha for their comments on earlier drafts of this paper. This research was sponsored partially by NASA under Grant NAS5–31351, by the National Science Foundation under Grant IRI–8719458, by the Air Force Office of Scientific Research Grant AFOSR–89–0303, and by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

References

[AB86] J. Archibald and J.L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM–Transactions on Computer Systems*, 4(4), November 1986.

[ABGM90] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM–Transactions on Database Systems*, 15(3):359–384, September 1990.

[ACL87] R. Agrawal, M. Carey, and M. Livny. Models for Studing Concurrency Control Performance: Alternatives and Implications. *ACM–Transactions on Database Systems*, 12(4):609–654, December 1987.

[BE89] H. Boral and P. Faudemay Edts. *Database Machines*. Springer–Verlag, June 1989.

[BS88] A. Bhide and M. Stonebraker. An Analysis of Three Transactions Processing Architectures. In *Proceedings of the 14th Very Large Data Base Conference*, pages 339–350, Los Angeles, CA, 1988.

[CD85] H.T. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 174–188, 1985.

[CFLS91] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client–Server DBMS Architecture. In *ACM–SIGMOD–Conference on the Management of Data*, Denver, CO, May 1991.

[DBP88] M. Darnovsky, J. Bowman, and K. Paulsel. *SYBASE Commands Reference*. Sybase, Inc., Emeryville, CA 94608, 1988. Document 3240–4.0.

[DGK+86] D. DeWitt, G. Graefe, K. Kumar, R. Gerber, M. Heytens, and M. Maralikrihsna. GAMMA–A High Performance Backend Database Machine. In *Proceedings of the 12th Conference on Very Large Data Bases*, Kyoto, Japan, August 1986.

[DMFV90] D. DeWitt, D. Maier, P. Futtersack, and F. Velez. A Study of Three Alternative Workstation–Server Architectures for Object–Oriented Database Systems. In *Proceedings of the 16th Very Large Data Bases Conference*, pages 107–121, Brisbane, Australia, 1990.

[DR91a] A. Delis and N. Roussopoulos. Performance Comparison of Three Modern DBMS Architectures. Technical report, University of Maryland, College Park, MD, 20742, May 1991. CS–TR–2679, UMIACS–TR–91–75.

[DR91b] A. Delis and N. Roussopoulos. Server Based Information Retrieval Systems Under Light Update Loads. In *Proceedings of the 1991 IEEE International Conference on Systems, Man, and Cybernetics*, Charlottsville, VA, October 1991.

[FD90] P. Fortier and G. Desrochers. *Modeling and Analysis of Local Area Networks*. CRC–Press, 1990.

[Fer84] D. Ferrari. On the Foundations of Artificial Workload Design. In *Proceedings of ACM–SIGMETRICS*, 1984.

[GLPT76] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. *Granularity of Locks and Degrees of Consistency in a Shared Database*. North–Holland, 1976.

[Hal89] G. Hallmark. Function Request Shipping in a Database Machine Environment. In H. Boral and P. Faudemay, editors, *Sixth International Workshop on Database Machines*, June 1989.

[HF86] R. Hagman and D. Ferrari. Performance analysis of several back–end database architectures. *ACM–Transactions on Database Systems*, 11(1):1–26, March 1986.

[KDG87] K. Kuspert, P. Dadam, and J. Gunauer. Cooperative Object Buffer Management in the Advanced Information Management Prototype. In *Proceedings of the 13th Very Large Data Bases Conference*, Brighton, UK, 1987.

[KGBW90] W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the Orion Next–Generation Database System. *IEEE–Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[PGK88] D.A. Patterson, G. Gibson, and R.H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 109–116, Chicago, IL, June 1988.

[PJ91] J. Purtilo and P. Jalote. An Environment for Developing Fault–Tolerant Software. *Transactions on Software Engineering*, 17(1991):153–159, February 1991.

[RD91] N. Roussopoulos and A. Delis. Modern Client–Server DBMS Architectures. *ACM–Sigmod Record*, September 1991.

[RK86] N. Roussopoulos and H. Kang. Principles and Techniques in the Design of ADMS+-. *Computer*, 19(12), December 1986.

[RKC87] W. Rubenstein, M. Kubicar, and R. Cattell. Benchmarking simple database operations. In *ACM–SIGMOD–Conference on the Management of Data*, pages 387–394, 1987.

[Rou91] N. Roussopoulos. The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis. *ACM–Transactions on Database Systems*, 16(3):535–563, September 1991.

[SKK+90] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE–Transactions on Computers*, 39(4), April 1990.

[SM86] K. Salem and Hector G. Molina. Disk Striping. In *Proceedings of the 1986 International Conference on Data Engineering*, Los Angeles, CA, 1986. IEEE.

[Ste90] R. Stevens. *Unix Networking Programming*. Prentice Hall, 1990.

[TR91] A. Thomasian and I.K. Ryu. The Analysis of Two Phase Locking Protocols. *Transactions on Software Engineering*, 17(5), May 1991.

[WN90] K. Wilkinson and M.A. Neimat. Maintaining Consistency of Client–Cached Data. In *Proceedings of International Conference on Very large Data Bases*, Brisbane, Australia, August 1990.

[Wol89] J. Wolf. The Placement Optimization Program: A Practical Solution to the Disk File Assignment Problem. *Performance Evaluation Review*, 17(1), May 1989.

[WR91] Y. Wang and L. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *Proccedings of the 1991 ACM SIGMOD International Conference*, Denver, CO, May 1991.