TANDEM

Guardian 90: A Distributed Operating System Optimized Simultaneously for High-Performance OLTP, Parallelized Batch/Query, and Mixed Workloads

Andrea Borr

Technical Report 90.8
July 1990
Part Number: 49788

Guardian 90: A Distributed Operating System Optimized Simultaneously for High-Performance OLTP, Parallelized Batch/Query, and Mixed Workloads

Andrea Borr
Tandem Computers Incorporated
Vallco Parkway
Cupertino, California 95014

ABSTRACT

The Tandem NonStop is a loosely-coupled multi-computer system managed by *Guardian 90*, a message-based distributed operating system designed to provide an environment for online transaction processing. One of the benefits of a loosely-coupled architecture is its inherently distributed character. A distributed architecture allows many components to be applied scalably in parallel to a large data-intensive task, such as batch or query processing, or to many small independent tasks, such as online transaction processing. However, achieving good performance in a loosely-coupled architecture presents some challenges relative to a tightly-coupled architecture. These challenges include:

- Overcoming the high cost of inter-process communication.
- Performing load-balancing without shared memory.
- Solving the client-server priority inversion problem.

Overcoming these challenges has required complex performance-oriented optimizations in Guardian. These optimizations -- aimed at reducing message traffic, avoiding software bottlenecks, achieving decentralized load-balancing, and avoiding client-server priority inversion -- are necessary in order to reap the potential benefits of parallelism inherent in a multi-computer architecture. This paper describes how Guardian achieves these goals. The price for their achievement is, however, the integration into Guardian of high-level database and TP-Monitor function. Such features are conventionally supported as layers on top of shared-memory-based operating systems.

CONTENTS

1. INTRODUCTION

1.1. System Design Goals

Design goals for the original Tandem NonStop System in 1974 were the provision of an online transaction processing (OLTP) architecture exhibiting fault-tolerance, high availability, and modular expandability [BART78]. These goals were achieved based on a loosely-coupled system architecture. With the achievement of the initial goals, focus shifted to fuller exploitation of other benefits of loose-coupling. These are the potential for *OLTP distribution* (both of data and execution) and *scalable parallelism*-- both natural fall-outs of loose-coupling that were originally given less emphasis.

The original implementation supported distributed database access to the extent that read-only queries spanning disks on separate processors or nodes could be executed with location transparency. Exploiting the architecture's potential for true OLTP distribution, however, had to await the addition to Guardian of support for distributed transaction management. The Transaction Monitoring Facility (TMF), introduced in 1981, enabled the atomic commitment of database updates by distributed cooperating processes working on the same transaction in multiple processors of a node, cluster, or network. TMF introduced to the Tandem system the *transaction* concept with the following properties, encapsulated by the *ACID* acronym [HAER83]:

Atomicity:	all or none of the transaction's effects appear in the database.
Consistency:	the transaction is a legal transformation of the database according to application-defined protocols.
Isolation:	transactions are isolated from seeing changes performed by concurrent transactions.
Durability:	once a transaction is committed, its changes persist, even in the event of later system faults.

The addition of TMF made the development of distributed database OLTP applications practical [BORR81].

The architecture supports OLTP scalability through *inter-transaction* parallelism: the simultaneous execution of multiple independent transactions on separate modules. Scalability of OLTP workloads means near-linearity in the curve representing throughput (transactions per second) vs. system and problem size. It does not, however, imply a reduction in individual transaction execution times (i.e. response times) as components are added to the system.

Execution time scalability for batch, query, and utility jobs, on the other hand, is achieved through *intra-job* parallelism. Transparent support for intra-job parallelism was added in 1989 with Release 2 of NonStop SQL, Tandem's distributed implementation of ANSI/ISO Structured Query Language. Release 2 automatically and transparently generates query plans employing parallelism within single SQL statements (i.e. intra-statement parallelism), and provides increased use of parallelism in utilities. Use of intra-job parallelism provides batch job execution scalability: near-linearity in the curve representing job execution time vs. system size [ENGL89]. Release 2 also introduced parallel index maintenance. Use of parallelism in multi-index maintenance helps OLTP response time by eliminating the penalty for the multi-index vs. the single-index case. By contrast, serial update of multiple indices occasioned by row insert, update, or delete may cause response time to grow in proportion to the number of indices.

1.2. Overview of System Configuration

The Tandem NonStop System is a "shared-nothing" multi-computer [STON86] [BHIDE88]: neither memory nor peripheral storage is shared among the processors. A Tandem system (node) consists of up to 16 loosely-coupled (non shared-memory) processors interconnected by dual high-speed buses. Nodes can be connected into clusters via fiber optic links, as well as into long-haul networks via X.25, SNA, or other protocols.

Disks (and peripherals in general) are physically connected to exactly two processors in a node and are accessed via an *IO-server* which runs as a fault-tolerant *process-pair* in those two processors. At any point in time, one processor contains the "primary" IO-server process controlling the active path to the device. The "backup" IO-server process resides in the other processor connected to the device. (Each processor contains a combination of primary and backup IO-servers, in addition to other system and application processes). The backup IO-server acts as a "hot-standby", providing fault-tolerance and device availability in the event of a single component (hardware or software) failure. All IO -- whether at system or application level – occurs via the client-server paradigm.

1.3. Overview of Software Structure

The resources of a Tandem NonStop System are managed by *Guardian 90*, a message-based operating system that hides the absence of shared memory by providing a uniform request-reply *Message System* interface for communication between processes executing in the same or different processors. The Message System makes the distribution of hardware components transparent [BART78]. The Message System is a proprietary design optimized for the request-reply model. It is CPU-intensive and consists of two layers: a request-response model layer and an interprocessor bus protocol layer. Guardian is structured as a system of cooperating processes that communicate as client-server using the Message System request-reply model.

The *File System* is an application programming interface to the Message System and to the IO-servers. When the client invokes File System procedures such as READ or WRITE, the File System formats and sends messages amongst application processes, IO-servers, and system processes. The File System makes location-transparent device IO and inter-process communication (IPC) available to system and application processes via a paradigm resembling *remote procedure call (RPC).* The Message System and the File System make the multiple-computer structure transparent, effectively transforming it into a single system image at the user level.

Guardian's message-based structure allows it to exert decentralized control over a local network of processors. Since it already addresses some of the problems of controlling a distributed computing system, Guardian has a natural extension to support a data communications network of Tandem nodes, each node containing up to 16 processors. The extension of Guardian to the network operating system, a product known as Expand, involves the generalization of message destinations to include processes in other nodes. This extension of the Message System beyond the boundaries of a single system allows a process anywhere in the network to send or receive a message from any other process in the network.

A *Disk Process* is an IO-server that manages a disk *volume* (optionally replicated on *mirrored* physical drives for fault tolerance). The disk "process" is actually an IO-server process group consisting of up to 8 (typically 3) cooperating processes, configured statically at system generation time. These processes run in the same CPU and share a common message input queue and a common area of memory holding control blocks and disk cache buffers. Each process is "single-threaded", having its own data space where its private execution stack is stored. The "backup" disk "process" (also a process group) runs as a "hot-standby" for fault tolerance in the second processor physically connected to the disk (see Section 1.2).

Each Disk Process performs functions ranging from low-level device driver functions to database record access, concurrency control, logging, transaction recovery, and the provision of high-availability and single-fault tolerance (termed NonStop by Tandem). The Disk Process implements high-level data management functions and yet must be viewed as part of the Guardian operating system.

For example, Guardian and the Disk Process are closely integrated in the support of the memory management function. One of the Disk Process' clients is Guardian's *Memory Manager*, a system process residing in each CPU that performs virtual-memory paging for its CPU by issuing *RPC* READs and WRITEs to the Disk Processes managing the disks (possibly attached to processors separate from the Memory Manager's) containing code files and virtual-memory swap files. Furthermore, the Disk Process' database buffer-pool management algorithms are integrated with Guardian's processor-global virtual-memory management mechanism [BORR88].

2. THE BENEFITS AND CHALLENGES OF LOOSE-COUPLING

This section defines loose-coupling and enumerates some of its advantages. Also identified are several obstacles to overcome if the advantages of loose-coupling are to be exploited.

2.1. Terminology

The following discussion focuses on the relative advantages and disadvantages of two architectures supporting high-performance transaction processing. Both architectures allow system expansion by adding components such as processors, disks, communication lines, etc. [HIGHL89].

Tightly-Coupled Multi-Processor: two or more processors connected to a common memory.

Loosely-Coupled Multi-Computer: two or more computer systems (each with its own memory) connected by a high-speed bus used primarily to pass messages between processes.

The *tightly-coupled* architecture corresponds to Stonebraker's category *shared memory* [STON86]. The coupling is termed "tight" because it is at the data level; that is, processes executing on different processors communicate by sharing common data in memory.

The *loosely-coupled* architecture corresponds to Stonebraker's category *shared nothing*. The coupling is termed "loose" because it is at the message level; that is, processes executing on different processors communicate by exchanging messages.

2.2. Inherent Distribution and Scalability are Benefits

One of the benefits of loose-coupling is its inherent distribution. A distributed architecture allows many components to be applied scalably in parallel to a large data-intensive task, such as batch or query processing, or to many small independent tasks, such as online transaction processing. The ability to configure the system without hardware bottlenecks, and the opportunity to avoid software bottlenecks, make it possible to achieve near-linear throughput and response time scalability with the addition of components.

One of the reasons that tightly-coupled systems have problems with scalability is that they require complex hardware to handle multiple access paths to the shared memory. Tightly-coupled architectures have historically realized sub-linear scalability due to memory access interference.

Logical contention is a more significant problem for the scalability of tightly-coupled systems. When data structures in memory are shared among processors, serialization of the processors is required when these data structures undergo modification. This tends to cause sub-linear scalability of the system.

By contrast, a loosely-coupled configuration does not suffer from memory access interference. Moreover, an important source of logical contention -- shared-memory data structures accessed by multiple processors -- is absent from loosely-coupled systems. Requirements for serial execution of the processors are minimal. Loose coupling therefore offers the potential for near-linear scalability with the addition of processors.

2.3. Reliability, Availability, Fault-tolerance are Benefits

System reliability -- and hence availability -- in a loosely-coupled system is enhanced by superior fault containment. It is possible to avoid globally shared modules (hardware or software) and hence "single points of failure". By contrast, one sick processor or process in a tightly-coupled system can contaminate critical portions of the shared memory. Furthermore, the modular nature of a loosely-coupled system facilitates on-line repair or replacement of modules, adding to the potential for high availability.

Hardware modular redundancy can be used even in tightly-coupled systems to increase hardware fault-tolerance. As reported in [GRAY85], however, analysis of the failure statistics of a commercially available computer system show that software and administration faults are more major contributors to outage than hardware faults. Only loose-coupling allows software fault tolerance. Tandem's implementation of software modular redundancy using process-pairs for critical system components has been shown to greatly enhance system availability [GRAY85]. Furthermore, a message-based operating system facilitates use of process-pairs through the client-server paradigm (discussed in sections 1.2 and 1.3).

2.4. Cost of Inter-Process Communication via Messages

The primary disadvantage of loose-coupling is that the lack of shared memory mandates a message-based operating system. Communication in a loosely-coupled system is based on exchanging messages between processes rather than on sharing memory. Even an optimized path through Guardian's Message System requires at least 2,500 instructions per request-reply circuit [UREN86]. By contrast, a tightly-coupled system can use shared memory parameter passing to exchange data between processes in instruction counts on the order of 25. This hundredfold penalty for loosely-coupled versus shared-memory communication is the main cost of a loosely-coupled design [BHIDE88].

2.5. Difficulty of Load-Balancing

Another challenge for loosely-coupled systems is the difficulty of load-balancing in the absence of shared memory for storage of work queues and process images accessible to all processors. This presents difficulties both in the scheduling of processes on CPUs and in the scheduling of access to services provided by a limited number of identical server processes spanning multiple processors.

In a tightly-coupled system, all processors can operate from a single dispatchable task queue (the *Run-Queue*) maintained in shared memory. For example, the ready-to-run process at the head of the queue is processed by the first available processor. It runs on this processor until it blocks (or is preempted). Upon becoming dispatchable once again, the process is re-queued in the Run-Queue. When it reaches the head of the queue, it is processed by the next available processor. Thus, a process may be passed amongst the tightly-coupled processors with each "ready-blocked-ready" cycle in its execution history. This tends to keep all processors busy as long as there is work to do.

In a loosely-coupled system, the lack of shared memory makes the implementation of the equivalent of a global Run-Queue impractical. Unlike the tightly-coupled case described above, a running process cannot easily be passed from processor to processor in order to keep all processors busy. While process migration from a heavily-loaded to a lightly-loaded processor may be an option, algorithms to accomplish transparent and dynamic load balancing in a distributed system using migration are still a research topic [HAC87].

These considerations make a careful choice of processor expedient when initiating a new process in a loosely-coupled system. If the goal is to keep all processors of the loosely-coupled complex equally busy, and if all submitted work items are assumed to have the same expected CPU cost, then the new process initiator should seek to place the new process on the least busy processor. However, since resources other than computational are distributed across the loosely-coupled complex, the proper criterion for process placement may relate more to proximity to a resource than to attempting to keep the processors equally busy.

While load balancing in distributed systems is a popular current research topic, researchers are concentrating on initial process placement and on process migration, where a process is implicitly viewed as representing a unit of work much more substantial and longer lived than the typical online transaction [HAC89]. The load-balancing problems encountered in the support of high-performance OLTP are different, however. Support for high-performance transaction processing would be difficult to achieve if the servicing of every submitted transaction required the overhead of new process initiation.

A solution to the support of high-performance OLTP is to deploy *server-classes* -- fleets of processes executing the same application or system service program. The servers run (semi-permanently) in multiple CPUs of a loosely-coupled complex. Transaction-Processing Monitors, "TP-Monitors", multiplex large numbers of clients onto members of relatively small server-classes. Examples of TP-Monitors are IBM's IMS, Tandem's Pathway, and Digital's ACMS.

The goal of server-class load-balancing is to minimize response time variance for a multitude of clients using a small number of highly-utilized servers. On one level, the goal is to provide uniform utilization and response time across the currently-running servers by causing each request to be routed to the server most likely to process it first. On another level, the goal is to adapt the running server class configuration to changes in load. This may mean expanding the server-class in one CPU, contracting it in another CPU, or migrating servers from busy CPUs to less highly-utilized CPUs.

Scheduling transaction requests by sending them to servers in a relatively idle CPU may not be the best strategy for clients seeking optimal response time. Suppose, for example, that a 2-member server-class is split between CPUs 1 and 2, with CPU 1 containing the server S1 and CPU 2 containing the server S2. Suppose that S1 is "busy" in the sense that it is currently servicing a client, but that it is blocked waiting for IO. Thus CPU 1 may in fact be idle, with its server-class subset blocked but nonetheless "busy". On the other hand, server S2 running in CPU 2 may be idle, though CPU 2 itself is highly utilized by other processes. A client of this server-class might get faster service from the idle server S2 in the highly-utilized CPU 2 than it would get from the "busy" server S1 in CPU1.

Managing server-classes and distributing the workload across them can present a challenge in the absence of shared memory. In a tightly-coupled system, shared memory can be used to hold queues of work waiting for service by a member of a server-class. By allowing the efficient implementation of a single queue per service, a shared-memory system can achieve optimal average response time for a system of identical servers, since a single queue allows better server utilization than that obtainable using multiple queues per service. The lack of shared memory in a loosely-coupled system makes the implementation of a single wait queue per service spanning multiple processors unacceptably expensive in terms of message and system coordination overhead. Guardian's present approach to

coping with the server-class load-balancing problem, as well as its evolutionary direction toward a better solution in the future through the integration into Guardian of TP-Monitor function, are described in Sections 3.22 and 3.23 respectively.

2.6. The Client-Server Priority Inversion Problem

Guardian has a preemptive priority scheduler and provides a user interface for setting relative process execution priorities as a means of controling process scheduling in each CPU. These features allow the user, by running response-time-critical OLTP applications at high priority and "batch" at low priority, to ensure that a high-priority process will be scheduled on the CPU ahead of a low-priority process.

Priority scheduling is, however, subject to an anomaly known as *priority inversion*, in which a process is delayed by the actions of a lower-priority process. One form, *client-server priority inversion*, is inherent in the client-server model. It potentially occurs when a low-priority client uses the services of a high-priority server.

The server in a client-server architecture typically runs at a higher priority than the client because the service cycle ties up scarce resources, including the server itself. The service cycle is performed at high priority in order to ensure that these resources are held on behalf of the client for the shortest possible time. While a high-priority server is servicing a client, however, the client's priority is effectively raised to that of the server with respect to the utilization of resources consumed by the service cycle, such as CPU, peripherals, and shared data structures. If the service cycle allocates a scarce resource to a low-priority client while a higher-priority process waits for the resource, priority inversion results.

To prevent priority inversion amongst enqueued requests from clients of differing priorities, the server can use *priority queueing*, in which enqueued requests are always serviced in priority order. This prevents initiation of service to a low-priority client when requests from high-priority clients are waiting. However, use of priority queueing does not address the following potential sources of priority inversion arising in the client-server environment:

- A high-priority server running on behalf of a low-priority client can potentially monopolize the server's CPU, to the disadvantage of other processes in that CPU with priorities greater than the client's but less than the server's. The problem is exacerbated if the service cycle is CPU-intensive.

- Service to requests from high-priority clients arriving at the server while it is busy on behalf of a low-priority client will be delayed until the current service cycle is complete.

Although both problems are worsened if the service cycle is long, a portion of the first problem is independent of service cycle length. Solving this portion of the problem requires a mechanism to cause the server to postpone initiating service to a low-priority client while the server's CPU has a high-priority process which is ready-to-run. Furthermore, preventing a long service cycle from aggravating the first problem requires a mechanism to preempt lengthy service to a low-priority client if a high-priority process becomes ready-to-run. Similarly, addressing the second problem requires a mechanism to preempt lengthy service to a low-priority client if a request from a high-priority client is enqueued for the server.

Both problems are worsened if a low-priority client is able to inundate a server with frequent requests. Even if the server uses priority queueing, it is still possible for a low-priority client to monopolize a server -- and, in the CPU-intensive service cycle case, the server's CPU -- with frequently issued requests which arrive at times when no higher-priority requests are enqueued for the server.

Monopoly of a server by frequent requests from a low-priority client is facilitated in loosely-coupled architectures such as Tandem's, where client and server may run in separate

CPUs. A low-priority client running in a lightly-loaded CPU might be able to execute frequently enough to inundate a high-priority server in another CPU with requests for CPU-intensive service. This may delay the execution of a ready-to-execute process, P, in the server's CPU whose priority is greater than the client's but less than the server's. Note that if process P had resided instead in the client's CPU, priority scheduling of the client process relative to process P would have prevented the client from executing so frequently.

A theoretical solution to the above problem would be to implement shared CPU scheduling via a single priority-ordered Run-Queue of dispatchable processes spanning the whole system. This would prevent a low-priority client in one CPU from issuing requests to a server in another CPU while higher-priority processes in the server's CPU are ready-to-execute. The global Run-Queue approach is not viable in a loosely-coupled architecture, however. The lack of shared memory makes its implementation unacceptably expensive in terms of message and system coordination overhead.

Client-server priority inversion is most pronounced when the server runs at a very high priority and performs a CPU-intensive service. The Disk Process (see Section 1.3) is a case in point. Although it uses priority queueing, the Disk Process is a potential source of client-server priority inversion. In fact, the only instance of client-server priority inversion observed in Guardian (excepting a few patholgical situations) has been in the interaction between the Disk Process and its clients.

The Disk Process runs at a very high execution priority for reasons explained in the following paragraphs. Furthermore, it may become CPU-bound during SQL "scans" because sequential read-ahead asynchronously moves the data being scanned to memory buffers. During such scans, the Disk Process evaluates single-variable query selection predicates and projection criteria against a long stream of records read in clustering order. Depending on the query, a scan may be read-only or may perform in-memory updates or deletes of selected records. Pre-fetch typically relieves the Disk Process from the need to perform any physical reads during such a CPU-bound service cycle. Physical writes of any changed data are delayed for asynchronous execution by a "post-write" mechanism.

One reason for the Disk Process' high priority is that Guardian's Memory Manager process (see Section 1.3) is one of its clients. The Disk Process must clearly service the Memory Manager at very high priority in order to avoid system deadlock.

Furthermore the Disk Process -- a statically-configured IO-server group consisting typically of 3 processes -- is a scarce resource. While it might seem that allowing dynamic growth of the process group would make it less of a scarce resource, in practice it turns out that growing the group beyond a small number of processes results in diminishing benefit because the scarce resources are actually the CPU, the semaphores protecting shared data structures, and the semaphore protecting the disk arm.

Thus, the Disk Process, as a scarce resource, would constitute a system bottleneck unless its service cycle processing were expedited at top priority, regardless of the priority of the client. As a consequence, the Disk Process service cycle can give rise to priority inversion in the absence of counter-measures. Guardian's approach to the solution to this problem is described in Section 3.3.

3. HOW GUARDIAN OVERCOMES THE CHALLENGES OF LOOSE-COUPLING

Section 2 pointed out three major challenges presented by loose-coupling. This section outlines the techniques and optimizations used in Guardian to address these challenges.

3.1. Message Traffic Reduction by Integrating SQL with Disk Process

Distributed DBMS implementations typically have a client-server structure. User interfaces are provided by frontend client process(es), which run on the user's processor or node. Actual disk IOs are performed by the DBMS server "engine" that runs on the processor or node physically connected to the disk. The partitioning of record access and transaction management function between client and server varies with the implementation.

Since minimizing client-server message traffic is a key performance issue for distributed DBMSs, the effect on message traffic should be considered when partitioning functionality between client and server. In general, message traffic between client and server can be minimized by managing shared resources (buffers, locks, file structures) as much as possible on the server side. Another important technique for reducing message traffic is to push projection and selection function "downward" to the server side. Thus, the server filters data being scanned (or manipulated) on behalf of the client, only returning to the client (or manipulating) data satisfying the client's predicate. This message-traffic saving technique is often characterized as "shipping function to the data" as opposed to "shipping data to the client."

Tandem's SQL implementation uses the above techniques to reduce client-server message traffic [BORR88]. This has required integration into Guardian's low-level disk IO system (i.e. the Disk Process) of such aspects of SQL semantics as the field-oriented interface and the set-oriented data manipulation operations of selection, update, and delete [ANSI]. By subcontracting SQL selection and projection logic to the Disk Process (wherever appropriate to the query execution plan), and by utilizing a field- and set-oriented interface with the Disk Process, NonStop SQL reduces message traffic between DBMS frontend client processes and Disk Processes. While knowledge of SQL semantics on the part of Guardian's disk IO subsystem may seem anomalous with respect to conventional system software layering, it is an example of the price paid in the Tandem architecture to achieve the optimized message traffic necessary for good performance in the loosely-coupled environment.

3.2. Guardian's Approach to Load-Balancing & Server-Class Management

The load-balancing problems of loosely-coupled systems were discussed in Section 2.5. This section discusses Guardian's present approaches to the problems of process placement and server-class load-balancing. Some limitations of the present approach to server-class load-balancing are discussed, and an approach which addresses the problem at a lower level of the system is described.

3.2.1. Selecting a Processor for New Process Initiation

In addition to providing a user interface for the placement of new processes on CPUs, the designers of Guardian chose to let users insert their own load-balancing algorithms into a "command interpreter monitor" process, CMON, which has an architected interface. CMON is automatically consulted when the user initiates a new process. The CMON logic has the option of placing the new process on a CPU according to a Tandem-supplied or user-coded algorithm. The policy could be either *static* or *adaptive*, depending on whether or not information about current system state is used [EAGER86].

3.2.2. Server-Class Load-Balancing: Non Guardian-Integrated Approach

Pathway, Tandem's TP-Monitor, is implemented on top of Guardian as a set of user-level processes. Pathway multiplexes a large number of clients onto a much smaller number of application server processes [PATHWAY]. A Pathway *requester* is a screen control and transaction flow program that accepts data from a terminal, workstation, or other device and sends one or more transaction request messages to arbitrary application *server* processes providing services needed by the transaction flow. A server typically accesses a database to satisfy a particular transaction request. After completing its work, the server returns its reply to the requester.

A Pathway server-class is a collection of identical application processes typically configured by the user to run in multiple CPUs, with multiple server copies per CPU. A server-class can consist of a *static* subset, initiated once at system startup, and a *dynamic* subset, initiated on an "as needed" basis and running only while the need exists. The size of the server-class, the preferred CPUs on which it is to run, and the size of the static and dynamic subsets are user-specified configuration parameters.

Each Pathway subsystem has a monitor process, PATHMON, that maintains a database of the configuration and status of the objects it controls, including the server-classes configured by the user. PATHMON implements distributed server-class load-balancing algorithms. By controlling the granting of the communication links between requesters and servers -- an activity known as *link management* -- the PATHMON algorithms implement a load-balancing policy that is transparent and *adaptive* in the sense that information about the current system state is used in certain situations.

PATHMON clearly cannot get involved in requester-to-server communication on a per-transaction basis without constituting a bottleneck. Rather, only on an exception basis do requesters ask PATHMON for a *link*, i.e. for permission to open communication to a member of a server-class. For example, the first time a requester needs the services of a particular server-class, PATHMON is asked for a link to that server-class. PATHMON replies with the name of a server in that class that can support another *open* from a requester. The maximum number of concurrent requester *opens* supported by each server in the server-class, MAXLINKS, is a configurable attribute of the class. If all the servers currently running already have *opens* from MAXLINKS requesters, then PATHMON may deny subsequent link requests, or else (depending on configuration) may decide to augment the server-class by starting some DYNAMIC servers.

Once the requester has obtained a link and has *opened* a member of the server-class, the resulting *open* session is long-lived. The requester then serially multiplexes a stream of transactions needing the same services on the set of server links it currently owns. If the requester is multi-threaded, and multiple concurrent transactions require services from the same server-class, then the requester obtains multiple links, as long as PATHMON grants them. When more links are denied by PATHMON, concurrent transactions from the same requester needing the same services must use the requester's existing links serially, possibly resulting in queueing.

3.2.3. Server-Class Load-Balancing: Guardian-Integrated Approach

PATHMON's load-balancing algorithms effectively implement a queue per server process when multiple requesters have access to a server-class that is configured with MAXLINKS greater than one. The disadvantage of such a queueing configuration is that there is the possibility of multiple requesters using their existing links simultaneously and queueing work for a busy server, unaware of the availability of idle servers in the same server-class. Setting the MAXLINKS attribute to one for the server-class eliminates this queueing behavior but expands the size of server-class needed to support the same number of requesters and lowers their average utilization.

Disregarding the potential for creating a bottleneck in the PATHMON process, one approach might be for PATHMON to act as a central clearinghouse and matchmaker between requesters and available servers on a per-transaction basis. Optimal server utilization could then be achieved by having the PATHMON process implement the effect of a single request queue per server-class. The message costs of such a *centralized control* approach to server-class load-balancing are clearly too high to support high-performance OLTP. In addition, it would not be a scalable architecture.

A *distributed control* approach to server-class load-balancing would, on the other hand, better satisfy the requirements of high-volume, high-performance OLTP. The fact that PATHMON is a user-level process, not integrated with the Guardian operating system, makes it difficult to evolve PATHMON's Link Management model into a distributed control mechanism with better queueing characteristics. Guardian is therefore in the process of evolving a distributed control approach involving integration of the server-class load-balancing function into its Message System and Process Control modules.

The postulated distributed control mechanism uses one queue per server-class per CPU in which the server-class runs. The servers of such a "CPU-server-subclass" can use shared-memory techniques, since they all run in the same CPU. Thus, the CPU-server-subclass servers share a common "new request queue". When an idle server in the CPU-server-subclass wants to pick up new work, it "listens" at the "new request queue" for work queued there by a requester for processing by an arbitrary server-class member. Thus, load-balancing within a CPU is automatically performed by the servers themselves using the shared queue, since a server will dequeue a request as soon as it is free to do so. This tends to minimize server process idle time.

With the CPU-server-subclass load-balancing problem thus resolved, the problem is now reduced to that of choosing a CPU from amongst the set of CPUs where the server-class runs. The approach will be heuristic, implemented with the aid of a per-CPU, "globally-updated" table called the Services Allocation Table (SAT).

The Message System already has an example of such a table, updated by broadcast to the (up to 16) CPUs of a node. The existing table, the Destination Control Table (DCT), maps a process "name" to its message queue address. The SAT will instead map a server-class or "service" name to the list of "new request queue" addresses of the CPU-server-subclasses. Also maintained in the SAT will be past performance statistics of the CPU-server-subclasses. Requesters will tend to target the CPU-server-subclasses in a round-robin fashion, trying to avoid those with a recent backlog problem, and complaining to a Guardian system process, the Service Manager, about lengthy waits for service. A requester timing out while waiting for service from a backlogged CPU-server-subclass might go on to try another CPU while the Service Manager augments the server-class with dynamic servers or possibly even migrates an entire CPU-server-subclass to another CPU.

The integration into the operating system of such TP-monitor function may seem anomalous with respect to conventional system software layering in a shared-memory system. On the other hand, such integration provides an opportunity for a distributed control solution to the server load-balancing problem in the loosely-coupled environment. The integrated server-class load-balancing mechanism represents a future direction for Guardian.

3.3. Priority Inversion Solution Facilitates Mixed Workload Handling

While client-server priority inversion could potentially constitute a generalized problem for Guardian, it has been observed only in the interactions of the Disk Process with its clients (aside from a few pathological instances). Hence, Guardian's solution has been focused on the Disk Process. The principles of the solution could easily be generalized to any high-priority server, however.

In addition to preventing Disk Process priority inversion, Guardian's solution provides an effective mechanism for using relative process priorities to allow concurrent processing -- with minimal degradation of OLTP throughput or response time -- of *mixed workloads* of high-priority, response-time-critical OLTP applications and low-priority, background batch or query processing. Support for mixed workloads requires that low-priority "batch" clients be prevented from using the services of the high-priority Disk Process to monopolize the CPU at the expense of high-priority OLTP jobs. It further requires that the Disk Process interrupt its processing of a low-priority SQL scan request if it detects a process competing for its CPU that is higher in priority than the SQL scan client, or if its input queue contains a request from a higher-priority client. The effectiveness of the solution in providing mixed workload support in a disk server architecture is independent of whether the underlying system uses shared memory or is loosely-coupled.

The solution has two parts. The first part consists of assuring that the priority of the client is taken into account when dispatching the Disk Process to initiate a client service cycle. This is partially accomplished by having the Disk Process use priority queueing; that is, it services its input queue in client priority order. Furthermore, Guardian tries to achieve the effect of a single Run-Queue spanning the CPUs of the Disk Process and its clients by having the Disk Process act as a surrogate for a remote client process on the Run-Queue of its own CPU. By scheduling the Disk Process in the Run-Queue at the priority of the head (i.e. highest priority) queued message (message priority is obtained from the priority of the client process), Guardian causes the Disk Process to refrain from initiating a client service cycle as long as the Run-Queue contains a process higher in priority than the client. If, while the Disk Process is waiting its turn in the Run-Queue at the client's priority, a higher-priority request is enqueued, or if an event occurs indicating high-priority work for the Disk Process, then Guardian will re-schedule the Disk Process in the Run-Queue at the priority of the new work. When it begins processing a request, the Disk Process resumes its usual high priority.

The second part of the solution consists of assuring that a lengthy, CPU-bound service cycle on behalf of a low-priority client does not delay service to a high-priority client, and does not deny use of the CPU to a high-priority ready-to-execute process. Thus, if servicing a client is expected to involve lengthy CPU-bound processing, the Disk Process allots an elapsed time QUANTUM to the service. During the service cycle, the Disk Process periodically "polls" for contention from higher priority work. If it detects contention -- either a high-priority queued message or a high-priority ready process -- the Disk Process practices *cooperative preemption* by either truncating or "slicing" the quantum.

For example, the Disk Process allots a quantum when processing a message requesting the execution of a portion of an SQL scan. When the quantum expires, the Disk Process replies to the client indicating that the scan is incomplete. The client then presents a "re-drive" request message to do the next portion of the scan. For each re-drive request message, the Disk Process does as much processing as it can on the scan -- limited by the quantum, as well as by the result of periodic preemption checks and by the reply buffer capacity (when data is being returned). The quantum can be interrupted by a process coming ready in the Disk Process' CPU whose priority exceeds that of the client, or it can be "sliced" in response to various contention conditions. When the quantum has expired or has been truncated, the Disk Process replies with a record place marker indicating how far it got. The client subsequently sends a new request message to "re-drive" the continuation of the scan starting at the record beyond the place marker.

The two parts of the solution thus ensure that Disk Process client priority -- relative to the priorities of other clients, as well as to the priorities of ready processes in the CPU -- is taken into account both when initiating service and when deciding to prolong service to a client. By making priority scheduling effective in the loosely-coupled client-server architecture, the solution provides Guardian with an effective mixed workload capability.

4. HOW GUARDIAN EXPLOITS THE ADVANTAGES OF LOOSE-COUPLING

This section describes some general techniques used in designs that apply loosely-coupled processors in parallel to achieve scalability. As examples of the exploitation of parallelism by Tandem system software, parallel query execution and parallel utility operation are outlined.

4.1. Avoiding Software Bottlenecks

One of the benefits of loose-coupling is that it affords the opportunity to avoid software bottlenecks in application and system software design. Two techniques especially suited to the avoidance of software bottlenecks in loosely-coupled systems are as follows:

- Use *distributed control* rather than centralized control solutions. An example of a distributed control solution is the postulated Guardian-integrated approach to server-class load-balancing, described in Section 3.23. Guardian currently uses distributed control algorithms in the Message System, Expand, and other subsystems.

- Use *partitioning* and *parallelism* to avoid "hot spots". An example of a curable "hot spot" on the database is the "end-of-file" point of an entry-sequenced table that is subject to a high volume of concurrent inserts. The table can be horizontally partitioned among disk volumes attached to multiple processors. This allows the inserts to proceed in parallel to multiple "end-of-partition" points, avoiding the high contention of a single "end-of-file" point. NonStop SQL uses this approach.

4.2. Using Parallelism in Designs that Scale Nearly Linearly

A job that is decomposable into independently executable sub-jobs lends itself to a design that subcontracts the sub-jobs to a set of loosely-coupled processors for parallel execution. A hoped-for benefit of using parallelism is *speedup*: linear reduction in the job's execution time as the number of subcontracted processors grows. Alternatively, the desired benefit may be linear *scaleup*: holding the execution time constant by adding processors in proportion to progressive increases in the job's size.

Designs that use parallel processors do not always approach linear scalability due to two categories of problems [SMITH89]:

- Startup, administration, skew.

- Interference.

4.2.1. Startup, Administration, Skew

Parallel processors take longer to start working on a job (e.g. due to the overhead of firing up sub-job execution processes on the participating processors). They furthermore suffer the overhead (typically messages) associated with sub-job administration. As the number of participating processors increases, the size of sub-job subcontracted to each decreases. Eventually, a point is reached where sub-job execution time is dominated by the startup and administration overhead.

The startup, administration, and skew problems are unavoidable. The skew problem, in particular, inevitably leads to loss of linearity due to diminishing returns associated with progressively finer task partitioning and growing numbers of subcontracted processors [SMITH89].

4.2.2. Interference

The processors working on a problem may interfere with one another or may queue behind some bottleneck. In contrast to the previous problems, however, the interference effect is more amenable to solution in a loosely-coupled architecture than in a tightly-coupled architecture. The benefit of loose-coupling over tight-coupling is the ability to design for a minimum of interference using, for example, the techniques discussed in the Section 4.1.

4.3. Automatically Parallelized Batch/Query In NonStop SQL

NonStop SQL exploits the parallelism of Tandem's loosely-coupled architecture by transparently and automatically implementing parallelism within an SQL statement. The query optimizer automatically detects opportunities for parallelism within single SQL statements. Where possible, it decomposes the relational operators -select, project, aggregate, join, insert, update, and delete -- into smaller jobs that can be executed independently on multiple processors and disks. The *intra-statement parallelism* thus achieved gives "batch" applications the ability to execute transparently in parallel on multiple independent processors and disks.

Intra-statement parallelism allows near-linear speedup: query execution speeds increase almost linearly as processors and disks are added to the system. In addition, intra-statement parallelism can help jobs restricted to a fixed "batch window" by providing near-linear scaleup: if equipment is added in proportion to the progressive increase in size of a batch job, the elapsed processing time can be held constant [ENGL89].

NonStop SQL uses Tandem's distributed hardware architecture, Guardian's support for the client-server paradigm, and TMF's support for distributed transactions in its implementation of intra-statement parallelism. The distributed hardware architecture enables the horizontal partitioning of data across multiple disks and processors. The client-server paradigm allows the use of parallelism in the query executor process structure. TMF enables the atomic commitment of database updates by distributed cooperating processes working on the same transaction.

Tables and indices may be horizontally partitioned -- based on key ranges -- across disk volumes (i.e. across Disk Processes). This horizontal partitioning is transparent to application programs. The disks on which the partitions reside can span the processors of a node and the nodes of a network.

Parallel query execution uses a "master" executor process that spawns "slave" executor processes in partition-holding processors. The "master" subcontracts to the "slaves" pieces of relational operator execution pertinent to their partitions. The work of the entire executor process set is tied to a single transaction by TMF.

As a simple example of how linear speedup is achieved using parallelism, consider a query that scans a large table. The table could be stored on a single disk accessed by a single processor, or it could be equally partitioned across ten disks and processors. The speedup achievable by using a query plan that scans the ten partitions in parallel is potentially ten-to-one.

The corresponding linear scaleup example consists of a table, scannable in time T, which grows over time to ten times its former size. If the table was stored on a single disk before scaleup, and partitioned among ten disks and processors after scaleup, then use of a parallelized query plan could hold the scan time to T.

The generalization of this approach to most of the relational operators applied to partitioned tables is straightforward. In the case of join, however, existing partitioning (if any) might not be directly usable. Existing partitioning of one or both of the tables being joined is used to advantage if the query plan permits. If no useful key fields participate in the join, however,

then the tables are repartitioned among all local processors using a hash function. Parallelism can be used in accomplishing the repartitioning. When it is complete, the join has been divided into many small joins that can be processed in parallel.

4.4. Use of Parallelism In Sort and Data Management Utilities

As databases become increasingly large, the speedup and scaleup properties of the utilities used to manipulate them are an important manageability issue. The use of parallelism in such functions as sort, load, index build, dump, reorganize, etc., can make the difference between acceptable and unacceptable execution times.

As in the case of parallelized query execution, the approach taken by parallelized utilities is to divide a large task into many smaller independent ones that can be performed in parallel. Again, the techniques used depend on Tandem's distributed hardware architecture, Guardian's client-server model, and NonStop SQL's support for data partitioning across disks and processors. Some examples follow.

The parallel sort utility, FastSort, uses multiple processors and disks if available [TSUK86]. FastSort partitions the data to be sorted among multiple subsort processes. One process, called the Distributor-Collector, reads the input file and distributes the records among the subsorts. Each subsort sorts the data as it receives it. Since each subsort receives only a fraction of the total data, it spends significantly less time sorting than it would if it were sorting the total. Furthermore, the subsorts proceed in parallel. After all of the data has been read and sent to the subsorts, the Distributor-Collector reads the records back from the subsorts, merges them into the final sorted order, and writes them to the output file.

Parallel index creation uses a slightly different approach to parallelism. When a user creates an index on a partitioned base-table, separate processes are spawned to read each partition of the base-table. If the index being created is also partitioned, a "sort-write" process per partition is spawned. Each "sort-write" process sorts the data belonging in its partition and writes the sorted output to it. The process that reads the records decides in which output partition each record belongs, and directs the record to that partition. Once again, each read process reads only a fraction of the data, and each "sort-write" process sorts and writes only a fraction of the total data. If there are m base-table partitions and n index partitions, this results in $m \times n$ parallelism.

Even when utilities lack explicit support for parallelism -- load, dump, reorganize, etc. -- the ability to operate them independently on multiple partitions gives the user a means of using parallelism. For example, although no formal parallel load exists, a user can create a parallel load on a partitioned SQL table by running independent loads of the partitions in parallel. If the data is loaded from tape, the degree of parallelism is limited by the number of tape drives on the system and the number of partitions in the target table. The tapes should have the data in sorted order, and the tape boundaries should be partition boundaries. If the load is done from disk, the degree of parallelism is limited by the number of partitions in the target table. In either case, each instantiation of the load utility reads just the data required for the partition it is loading.

In the case of CPU-bound operations, such as sort and index creation, utilities that use a serial approach can suffer from an imbalance between their usage of IO bandwidth and CPU. Most of the IO bandwidth is wasted because of the CPU bottleneck resulting from the serial execution. On the other hand, the price paid for parallelization is the added overhead of moving data around to gain concurrency. This cost can be characterized by comparing CPU cost per record in the serial and parallel approaches. The added CPU cost of extra data movement in the parallel case is the price of a shorter elapsed time.

5. SUMMARY

Tandem's original design goals were to implement a fault-tolerant system supporting OLTP whose performance scales nearly linearly with the addition of components. The design used loose-coupling and a message-based operating system to achieve those goals. Over the years, Tandem software has been enhanced to take progressively more advantage of the parallel architecture to achieve scalability in batch, query, and utility operations as well as in transaction processing. The load control issues associated with support for mixed batch and OLTP environments have been successfully addressed. However, these achievements have necessitated first overcoming the challenges of loose-coupling -- costly inter-process communication, difficult load-balancing, and client-server priority inversion. Doing so has required implementing complex performance optimizations and bottleneck-avoidance strategies. In some cases it has required the integration of high-level function into low-level system services. This unconventional layering of system software allows Guardian to overcome the challenges of loose-coupling, a prerequisite to exploiting the parallel architecture.

6. ACKNOWLEDGEMENTS

Thanks are due to Diane Greene, Darrell High, Rich Larson, Pat Barnes, and Jim Gray for editorial suggestions. The sections on client-server priority inversion owe much to Susanne Englert, who helped design the algorithms, performed all measurements for the evolving solution, and analyzed mixed workload performance. Carol Pearson and Charles Levine were kind enough to contribute material on the implementation and performance of parallelized utilities.

7. REFERENCES

[ANSI] "Database Language SQL 2 (ANSI working draft)," ANSI X3H2 87-8. Dec. 1986.

[BART78] Bartlett, J. F., "A 'NonStop' Operating System," Proc. Eleventh Hawaii International Conference on System Sciences, 1978.

[BHIDE88] Bhide, A., "An Analysis of Three Transaction Processing Architectures," Proc. 14th International Conference on Very Large Data Bases, 1988.

[BORR81] Borr, A. J., "Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing," Proc. 7th International Conference on Very Large Data Bases, Sept. 1981.

[BORR88] Borr, A. J., and Putzolu, F., "High Performance SQL Through Low-Level System Integration," Proc. SIGMOD 88, ACM, June 1988

[EAGER86] Eager, D. L., Lazowska, E. D., and Zahorjan, J., "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," Performance Evaluation, Vol. 6, No. 1, March 1986.

[ENGL89] Englert, S. et. al., "A Benchmark of NonStop SQL Release Demonstrating Near-Linear Speedup and Scaleup on Large Databases," Tandem Technical Report 89.4, Cupertino, CA, May 1989.

[GRAY85] Gray, J., "Why Do Computers Stop and What Can Be Done About It?", Tandem Technical Report 85.7, Cupertino, CA, June 1985.

[HAC87] Hac, A., and Jin, X., "Dynamic Load Balancing in a Distributed System Using a Decentralized Algorithm," Proc. 7th IEEE International Conference on Distributed Computing Systems, Jan. 1987.

[HAC89] Hac, A., "Load Balancing in Distributed Systems: A Summary," Performance Evaluation Review, Vol. 16 #2-4, Feb. 1989.

[HAER83] Haerder, T., and Reuter, A., "Principles of Transaction-Oriented Database Recovery," ACM Computing Surveys, Vol 15.4, 1983.

[HIGHL89] Highleyman, W. H., "Performance Analysis of Transaction Processing Systems," Ch. 2, Prentice-Hall, 1989.

[PATHWAY] PATHWAY System Management Reference Manual, Tandem Computers Inc., Cupertino, CA, Part: 82365, 1985.

[SMITH89] Smith, M. et. al., "An Experiment on Response Time Scalability in Bubba," Proc. 6th International Workshop on Database Machines, June 1989.

[STON86] Stonebraker, M., "The Case for Shared Nothing," IEEE Database Engineering Bulletin, 9(1):4-9, March 1986.

[TSUK86] Tsukerman, A. et. al., "FastSort: An External Sort Using Parallel Processing," Tandem Technical Report 86.3, Cupertino, CA, May 1986.

[UREN86] Uren, S., "Message System Performance Tests," Tandem Systems Review, V2.3, Cupertino, CA, Dec. 1986.